The Hoard: Paged Virtual Memory for the Cell BE SPE

Jonathan Adamczewski, B. Comp. (Hons)

Submitted in fulfilment of the
requirements for the Degree of
Doctor of Philosophy

University of Tasmania

18th July, 2011

Note

This thesis should be printed in colour, or an additional copy requested from the author. Black and white reproduction (such as a photocopy) will lose meaning from the document.

Declaration of Originality

This thesis contains no material that has been accepted for a degree or diploma by the University or any other institution, except by way of background information and duly acknowledged in the thesis, and to the best of my knowledge and belief no material previously published or written by another person except where due acknowledgement is made in the text of the thesis, nor does the thesis contain any material that infringes copyright.

Jonathan Adamczewski

Authority of Access

Jonathan Adamczewski

Statement of Ethical Conduct

The research associated with this thesis abides by the international and Australian codes on human and animal experimentation, the guidelines by the Australian Government's Office of the Gene Technology Regulator and the rulings of the Safety, Ethics and Institutional Biosafety Committees of the University.

Jonathan Adamczewski

Abstract

The hoard is a paged virtual memory system for program data designed for the Cell Broadband Engine processor (Cell BE). The hoard makes it easier to utilise the performance available from the processor's Synergistic Processor Elements (SPEs). Traditional high-performance hardware cache designs are not well-suited to being implemented in software on the Cell BE architecture, so the hoard takes advantage of the SPE's wide registers to implement a multi-level page table. This permits fast, fully associative lookup into a dynamically-sized collection of data pages. Write-back of all pages is performed (not just those modified) and a first-in first-out page replacement method is used. This design is compared with the software cache provided with the IBM SDK for Multicore Acceleration — a fixed-size, four-way set associative design, using a traditional hardware approach — through extensive experiments with a diverse range of benchmarks. An in-depth analysis is made of the performance of the hoard design, addressing its strengths and shortcomings.

From this analysis, potential improvements to the hoard are considered, implemented, and their relative benefits measured. Changes include the tracking of page modification, pre-writing of pages, use of a write-through policy, pre- and partial-fetching of pages, and alternative replacement algorithms. Results show that simple and speculative methods are often the most beneficial, and that performance varies greatly depending on the memory access patterns of a particular program. It was found that programs with high locality of access will perform better when using the hoard, in some cases exceeding the performance of the same program running on the Power Processor Element of the Cell BE. For programs with lower locality of reference, performance of the hoard and SDK cache are within a few percent of one another.

An additional design goal for the hoard is to make the access and management of data outside of the SPE's local store as transparent as possible to the programmer. Not implemented as part of a particular compiler, the hoard is a compile- and run-time library that makes use of C++ language features to provide a natural pointer interface, requiring a minimal amount of changes to existing programs. Careful implementation of the interface ensures that program semantics are well preserved and accidental misuse is discouraged, while taking advantage of the features of the Cell BE to provide high-speed lookup. The result is a tool that facilitates the reuse of a broad range of existing code-bases on the SPE.

Acknowledgements

There are so many people that have contributed to this thesis, many without realising it.

Thanks to Arthur, for stepping forward at the very start to supervise a candidate lacking a clear direction, and to Julian for joining him in that task. There is no way that this would have made it to completion without your encouragement, detailed and thorough feedback, and honest reflections on my progress. Thanks also to Arthur for a name for the thesis.

The review and comments from Arthur, Julian, Ian, David, Joe, and Karen have made an immeasurable difference to this thesis. Thank you all for your persistence and effort.

My thanks to the staff of the School of Computing & (now) Information Systems who were especially helpful with the range of strange and awkward requests — Dave who could always find me a desk, an IP address, a TV, and/or a PS3 when needed, Andrew for great fortitude in repeatedly confronting the video-conferencing system, Tony, Christian, Julia, Andrea, and all the others who answered questions or solved problems over the course of writing this thesis.

Thanks to Ian Lewis for the very occasional, but wonderfully in-depth conversations, and some useful hints.

Thanks to the very helpful individuals at the Barcelona Supercomputing Center, who very generously provided access to their Cell BE blades.

To Jeremy Kerr for taking the time to drive me around Canberra and introduce me to OzLabs, Ben Elliston and others there (and in other IBM offices) who have answered questions and been generally encouraging in lots of little ways, not least Ken Werner, Ulrich Weigand, and Arnd Bergmann. And to those from (and formerly from) Sony who have also offered assistance on numerous occasions, especially Geoff Levand and Andrew Pinski. My sincere thanks to you all.

And thanks to the (other) resident lurkers of `#ps3dev`, `#ps3linux`, `#cell`, `#ouros` (forever tears), and `#cesspool`. All these channels have served to provide technical support, entertainment, distraction, sanity, conversation in obscure dialects, and — from time to time — very useful information. Thanks also to the many gamedevs (and other devs) for their banter and willingness to answer silly questions from time to time via Twitter.

Thanks to Menace and all the cr3w for making `dcypher` such a great way to lose entire weeks.

To Will and Gill for friendship, love and care, for me and my family, and to the Connections mob for being such a fantastically wonderful and diverse bunch.

Thanks to my parents and Karen's for their continued care and support for our family.

And many, many thanks to my loving, patient, supportive, hard-working, and encouraging wife and to my wonderful, wonderful daughters. I don't know what comes next, but I do know I'm looking forward to it.

Jonathan Adamczewski

Burnie, Tasmania, Australia

September 2010

Photo: Karen Adamczewski.

Contents

List of Figures

List of Tables

The Hoard: Paged Virtual Memory for the Cell BE SPE

Jonathan Adamczewski, B. Comp. (Hons)

Submitted in fulfilment of the
requirements for the Degree of
Doctor of Philosophy

University of Tasmania

18th July, 2011

The Hoard: Paged Virtual Memory for the Cell BE SPE

Jonathan Adamczewski, B. Comp. (Hons)

Submitted in fulfilment of the
requirements for the Degree of
Doctor of Philosophy

University of Tasmania

18th July, 2011

Note

This thesis should be printed in colour, or an additional copy requested from the author. Black and white reproduction (such as a photocopy) will lose meaning from the document.

Declaration of Originality

This thesis contains no material that has been accepted for a degree or diploma by the University or any other institution, except by way of background information and duly acknowledged in the thesis, and to the best of my knowledge and belief no material previously published or written by another person except where due acknowledgement is made in the text of the thesis, nor does the thesis contain any material that infringes copyright.

Jonathan Adamczewski

Authority of Access

Jonathan Adamczewski

Statement of Ethical Conduct

The research associated with this thesis abides by the international and Australian codes on human and animal experimentation, the guidelines by the Australian Government's Office of the Gene Technology Regulator and the rulings of the Safety, Ethics and Institutional Biosafety Committees of the University.

Jonathan Adamczewski

Abstract

The hoard is a paged virtual memory system for program data designed for the Cell Broadband Engine processor (Cell BE). The hoard makes it easier to utilise the performance available from the processor's Synergistic Processor Elements (SPEs). Traditional high-performance hardware cache designs are not well-suited to being implemented in software on the Cell BE architecture, so the hoard takes advantage of the SPE's wide registers to implement a multi-level page table. This permits fast, fully associative lookup into a dynamically-sized collection of data pages. Write-back of all pages is performed (not just those modified) and a first-in first-out page replacement method is used. This design is compared with the software cache provided with the IBM SDK for Multicore Acceleration — a fixed-size, four-way set associative design, using a traditional hardware approach — through extensive experiments with a diverse range of benchmarks. An in-depth analysis is made of the performance of the hoard design, addressing its strengths and shortcomings.

From this analysis, potential improvements to the hoard are considered, implemented, and their relative benefits measured. Changes include the tracking of page modification, pre-writing of pages, use of a write-through policy, pre- and partial-fetching of pages, and alternative replacement algorithms. Results show that simple and speculative methods are often the most beneficial, and that performance varies greatly depending on the memory access patterns of a particular program. It was found that programs with high locality of access will perform better when using the hoard, in some cases exceeding the performance of the same program running on the Power Processor Element of the Cell BE. For programs with lower locality of reference, performance of the hoard and SDK cache are within a few percent of one another.

An additional design goal for the hoard is to make the access and management of data outside of the SPE's local store as transparent as possible to the programmer. Not implemented as part of a particular compiler, the hoard is a compile- and run-time library that makes use of C++ language features to provide a natural pointer interface, requiring a minimal amount of changes to existing programs. Careful implementation of the interface ensures that program semantics are well preserved and accidental misuse is discouraged, while taking advantage of the features of the Cell BE to provide high-speed lookup. The result is a tool that facilitates the reuse of a broad range of existing code-bases on the SPE.

Acknowledgements

There are so many people that have contributed to this thesis, many without realising it.

Thanks to Arthur, for stepping forward at the very start to supervise a candidate lacking a clear direction, and to Julian for joining him in that task. There is no way that this would have made it to completion without your encouragement, detailed and thorough feedback, and honest reflections on my progress. Thanks also to Arthur for a name for the thesis.

The review and comments from Arthur, Julian, Ian, David, Joe, and Karen have made an immeasurable difference to this thesis. Thank you all for your persistence and effort.

My thanks to the staff of the School of Computing & (now) Information Systems who were especially helpful with the range of strange and awkward requests — Dave who could always find me a desk, an IP address, a TV, and/or a PS3 when needed, Andrew for great fortitude in repeatedly confronting the video-conferencing system, Tony, Christian, Julia, Andrea, and all the others who answered questions or solved problems over the course of writing this thesis.

Thanks to Ian Lewis for the very occasional, but wonderfully in-depth conversations, and some useful hints.

Thanks to the very helpful individuals at the Barcelona Supercomputing Center, who very generously provided access to their Cell BE blades.

To Jeremy Kerr for taking the time to drive me around Canberra and introduce me to OzLabs, Ben Elliston and others there (and in other IBM offices) who have answered questions and been generally encouraging in lots of little ways, not least Ken Werner, Ulrich Weigand, and Arnd Bergmann. And to those from (and formerly from) Sony who have also offered assistance on numerous occasions, especially Geoff Levand and Andrew Pinski. My sincere thanks to you all.

And thanks to the (other) resident lurkers of `#ps3dev`, `#ps3linux`, `#cell`, `#ouros` (forever tears), and `#cesspool`. All these channels have served to provide technical support, entertainment, distraction, sanity, conversation in obscure dialects, and — from time to time — very useful information. Thanks also to the many gamedevs (and other devs) for their banter and willingness to answer silly questions from time to time via Twitter.

Thanks to Menace and all the cr3w for making `dcypher` such a great way to lose entire weeks.

To Will and Gill for friendship, love and care, for me and my family, and to the Connections mob for being such a fantastically wonderful and diverse bunch.

Thanks to my parents and Karen's for their continued care and support for our family.

And many, many thanks to my loving, patient, supportive, hard-working, and encouraging wife and to my wonderful, wonderful daughters. I don't know what comes next, but I do know I'm looking forward to it.

Jonathan Adamczewski

Burnie, Tasmania, Australia

September 2010

Photo: Karen Adamczewski.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

cache
a collection of items of the same type stored in a hidden or inaccessible place

hoard

a stock or store of money or valued objects, typically one that is secret or carefully guarded
an amassed store of useful information, retained for future use

oxforddictionaries.com 2010

This research began with the idea of a software *cache* for the Cell BE SPE. It became apparent that 'cache' may not be the best word to use, as the assumptions that go into a traditional cache design were not all appropriate for the architecture of the SPE, and to call it a cache would evoke unhelpful associations in the mind of the user or researcher. The name *hoard* was suggested, and it has stuck.

This thesis documents the design and development of the hoard — a software paged virtual memory system for the SPE, designed to aid program portability requiring minimal changes to an existing program — and examines its performance for a selection of benchmark programs. The hoard simplifies the writing and reuse of programs for the SPE while remaining suitable for use in a broad range of program types, and is used through a C++ smart-pointer style interface.

From the initial analysis, potential improvements are devised and tested to yield a versatile, high-performing design.

The contents of this thesis are as follows:

Chapter 2 provides background and context for the ways memory hierarchies are commonly handled in computer systems, covering topics related to caching and virtual memory. Chapter 3 looks in depth at research related to the memory architecture of the SPE. Chapter 4 contains a deeper examination of the Cell BE hardware, particularly as it pertains to implementing a virtual memory system for the SPE.

The design of the hoard is presented in Chapter 5, from the initial ideas through to a completed implementation and the way it interacts with features of the Cell BE. In Chapter 6, the hoard API — the interface of the hoard exposed to the programmer — is described, with explanation of the problems encountered in its development and the solutions implemented to create a C++ smart-pointer class that is able to be safely used in place of a regular pointer. The chapter includes a case study of converting one of the programs benchmarked in this thesis to make use of the hoard, and considers some of the difficulties that may be encountered by a programmer using the hoard.

Chapter 7 contains the experimental method used to evaluate the hoard, along with the benchmark programs chosen for doing so. In Chapter 8 the results of this evaluation are presented with analysis. Based on these results, Chapter 9 details various attempted optimisations of the hoard, working towards a tailored hoard design and configuration that will perform well for a wide range of memory access patterns.

Chapter 10 provides the conclusions from this research and Chapter 11 lists further work that could be performed to extend this research.

Chapter 2

Background — Memory Hierarchies

> *"Ideally one would desire an indefinitely large memory capacity such that any particular . . . word would be immediately available . . . We are . . . forced to recognise the possibility of constructing a hierarchy of memories, each of which has greater capacity the preceding but which is less quickly accessible."* (Burks et al. 1946, p. 4)

In designing the hoard to meet the needs and use the features of the Cell BE SPE, understanding something of the history of how the same problems have been solved in different contexts is useful. This chapter gives an overview of that history, which informs the design of the hoard in later chapters.

Computer designs make use of memories of various sizes. Smaller memories are able to be accessed more quickly, where larger, slower memories allow the storage of more expansive datasets. In addition to speed and size, other constraints such as price, power, system performance, availability, and required functionality influence the decisions of system designers (Hennessy & Patterson 2007).

As per Tanenbaum (2007) and Hennessy & Patterson (2007), the memory hierarchy in a 'typical' computer consists of:

- A register file — a number of small storage locations that may be accessed quickly and directly by the processor.
- Caches — small, fast storage used to hold data that is likely to be needed soon by a running program.

- Primary storage — a series of locations for storing program text and data, able to be directly accessed by the processor.
- Secondary storage — a slower, larger storage system (disk, networked storage, etc.) that is not directly accessible by the processor, but that allows for a memory address space larger than the size of primary store.

The exact method of configuring and connecting these components varies, but they are the established basic building blocks of high performance systems.

When writing programs for a system with a hierarchy like this, software abstractions have evolved to the point where the average programmer is able to ignore all of these details. The vast majority of high-level programming languages take the assignment of instruction operands to processor registers away from the programmer, to be handled by the compiler or interpreter. Memory caches operate in a fashion that is transparent to the running program, locations for program data in primary storage are assigned by the operating system, and secondary storage will be utilised as necessary based on the demands of this and other programs running on the system, also managed by the operating system. It is not necessary for the average programmer to engage directly with any of these memory hierarchy details. Drepper (2007) provides an in-depth introduction to the ways in which each of these work in practice.

The abstraction commonly used to hide any the distinction between primary and secondary storage from a programmer is virtual memory (Denning 1970).

2.1 Virtual Memory

2.1.1 Overview

Virtual memory is, in general use, an abstraction that is provided by the hardware and operating system to hide the details of the physical memory architecture from a running program. In a system with virtual memory, the translation of virtual to physical addresses is not a detail that needs consideration within the logic of a program.

There are a number of benefits of virtual memory: it *decreases complexity* for the programmer, who can ignore the implementation details of the memory architecture, and provides *generality*, that programs may be written with the assumption of a single large flat

memory space and helps to make them easily portable across a range of systems.

Reduced program complexity and increased generality comes at a cost. Hardware and operating system routines for managing virtual memory require greater complexity, and incur a time penalty in their use.

In systems that lack virtual memory, the specifics of the memory architecture are exposed to the running program and, consequently, the programmer. The program must include explicit logic to track and transfer data between memory levels.

For systems without virtual memory, the use of overlays is one approach that has been used to help manage this process (Pankhurst 1968). The use of modules (or functions, or subroutines) and data through the course of a program's execution are statically examined and these components are mapped into the memory space of the program. These components are moved to and from physical memory locations at prescribed points during the execution of the program.

Such manual methods are time consuming to implement and add an extra degree of complexity for the programmer to manage. Programs that have highly indeterminate memory access patterns are more difficult to partition and efficiently manage (Denning 1970).

The extra work involved in manual memory partitioning is not without benefits. Clarifying memory access patterns does make it clear when data may be fetched in advance of its use, and written back as soon as it is finished with.

Originally developed as a way to hide the separation of storage devices on a system, virtual memory simplifies the labours of programmers. It was first implemented in the Atlas computer from Manchester University (Kilburn et al. 1962), and was also notably present in the commercial Burroughs 'large systems' of the early 1960s. The Burroughs systems provided a novel hardware implementation of virtual memory where data was accessed through descriptors (rather than the otherwise ubiquitous and typically unadorned 'pointer'). Descriptors provided a hardware supported mechanism for tracking the presence of data in main store or in secondary storage of disk or drum (Organick 1973).

Denning (1970) provides a detailed analysis of some of the benefits and penalties of using virtual memory, and makes a strong case for its inclusion in systems. Of particular note are the reduction in programmer time required and that performance is typically comparable.

It is also noteworthy that benefits and features of virtual memory extend beyond mem-

ory management. Denning (1996) mentions some other needs met through the use of virtual memory systems. These include memory protection which allows control over which parts of memory a program may access, what sort of access is possible (e.g. read only, read/write, no-execute), and program modularity (being able to combine separately compiled components into a program, with no requirement of explicit coordination of memory placement).

Virtual memory incurs an extra processing cost on a running system, requiring a mapping of virtual memory addresses to physical memory locations on every memory access. In addition to this per-access overhead, the virtual memory system is responsible for moving data into physical memory as required by the running program(s), and writing out data to make space as needed.

2.1.2 Implementation

Virtual memory is typically implemented using segmentation, paging, or a combination of the two (Denning 1970).

Segments are variably sized sections of memory containing code and/or data that a program may access while running. Segments provide a way to manage the various parts of memory to which a program requires access, in an exclusive or shared fashion. Segmentation provides an effective method for managing the logical separation of program resources. For virtual memory, segments alone have some limitations: managing the placement of variable-sized objects can be expensive in terms of time and space, particularly for multi-programmed systems; segments are limited to the size of memory; and the locality of access for programs is such that not all of a single segment is typically required in memory at the same time (Tanenbaum 2007).

Paged virtual memory is the division of the virtual address space into fixed-size pages. The advantages of this method are the easier management of data (re)placement in memory and typically greater efficiency in moving pages from storage to storage. With paging, the amount of space required and allocated will almost always be larger than the amount of memory that a program requires as a single page is the minimum effective size of an allocation. It is possible for the operating systems to help reduce this cost through the use of pooled memory allocations.

Paging is often combined with segmentation to provide flexibility, efficient manage-

ment of storage, memory protection and other advantages (Tanenbaum 2007). When combined in this way, a virtual address contains the necessary information to find the location of the particular information in a segment and page, consisting of (at least) the segment number, page number and an offset within a page.

2.1.3 But, at what cost?

The use of virtual memory incurs costs. These include performance, system complexity and efficiency of memory use.

The page table serves as the way to map pages to locations, and may be implemented in a number of ways, with various trade-offs in terms of size and space. Perhaps the simplest page table lists virtual addresses and their mapping to a physical location. While simple and fast to lookup the mapping for a given page, this method requires entries for all virtual pages and so will need to be large enough for all virtual pages of all processes in the system. Tanenbaum (2007) considers an example for a 64 bit system with 4KB pages and concludes that

> *"Tying up 30 million gigabytes just for the page table is not a good idea, not now and probably not next year either." (p. 200)*

Techniques that are used to reduce the space needed for page tables include:

- inverted page tables;
- hash lookups;
- larger page sizes; and
- multi-level page tables.

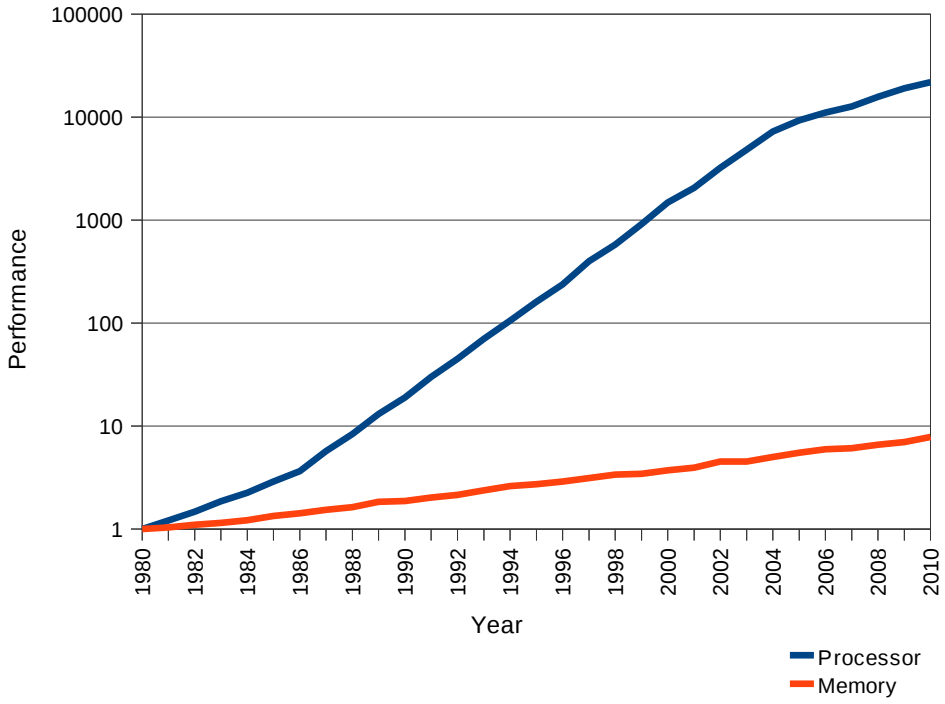
Inverted page tables map physical locations to virtual addresses and have the advantage of requiring entries only for physical memory, not the full size of the virtual address space. Additionally, they may be dynamically sized, containing only entries for mapped pages. Lookup requires a slower linear search, but the reverse page table is very space efficient.

Hash tables require more space than a minimal reverse page table, with the size being traded off to gain speed of access, but will provide closer to the constant time lookup of a simple page table.

Another way to reduce the storage required for page tables is to use larger page sizes. By doing so, the total number of pages needed for the address space of a program is reduced. With fewer pages there are fewer page translation entries required, regardless of the page table configuration. The cost of this approach is in decreased allocation efficiency as less space is used in some pages, and page-fault latency increases with the amount of data that must be transferred between memories. Page sizes of 4KB and 8KB are not uncommon, and many processors additionally offer huge page sizes for use with large datasets, e.g. 4MB on x86, 256MB on IA64, and up to 16GB on Power5+ systems.

Multi-level page tables are more complex to manage but reduce the size needed for page tables. To perform the translation from virtual address to physical location will require at least one extra memory access for each level of page table. Each of these is described in more detail in Tanenbaum 2007, p. 198–201.

CPU performance has increased at a far greater rate than memory performance, and the result has been a widening gap between the two, as is illustrated in Figure 2.1. By way of a current example, the Cell BE processor at 3.2GHz experiences a delay around 500 cycles to



Using 1980 performance as a baseline, this graph shows the widening gap between memory and CPU performance. Graph is based on the graph on page 289 of Hennessy & Patterson (2007).

Figure 2.1: The gap in performance between memory and processors over time

read from RAM[1], and a 3.8GHz Intel Core 2 Duo system[2] experiences around 400 cycles of latency[3].

It's clear from these numbers that accessing memory is time consuming and undesirable, but address translations occur on almost every memory access. These in turn require further accesses to memory to perform the translation. To speed up this process and to avoid multiplying the number of memory accesses that must occur, a translation lookaside buffer (TLB) is typically included in a processor to cache the most recently used mappings. It is the presence of a fast TLB that makes a reverse page table a useful option for lookup.

Analogous to a TLB, a segment lookaside buffer (SLB) may also be included in a processor to avoid memory accesses to map a virtual address to a particular segment.

2.1.4 Replacement

When a program requires a page that is not in primary store, a page fault occurs. Program execution will be interrupted while a handler locates and transfers the page from its location in secondary storage. This will typically take far longer than an access to primary storage (by 10–100 times). This being the case, it is desirable to seek to minimise the number of page faults that occur during the execution of a program.

Data access by a program is difficult or impossible to fully predict. There is a need for a page replacement system that will be able to make a best guess as to which pages will not be needed in the near future, so that pages may be replaced as infrequently as possible. For paged virtual memory, this may be achieved in a number of ways.

Well-written programs tend to exhibit a high locality of reference — access to particular pieces of data tend to happen close together in time, and data accesses tend to be to addresses close to one another. From this it can be concluded that if a page has been accessed recently, it is likely to be accessed again in the near future. The concept of locality and the history of the term is summarised neatly in Denning (2005).

At a given point in time, there will be a set of pages that will be accessed by a particular program, known as the working set. Due to the localised nature of references, a program's working set does not change much over a small time window. If the working set for a program is smaller than primary storage, page faults are much less frequent. From this

[1] Approximately 150ns, as measured with `dmabench` from the IBM SDK for Multicore Acceleration Version 3.1 (IBM Corporation et al. 2008c)

[2] upon which this thesis was authored

[3] Approximately 100ns, measured with the Later memory latency benchmark from http://later.sf.net

is can be seen that pages accessed within a certain time window are likely to be accessed again in the near future. Pages which are not accessed during that time window are less likely to be needed (Denning 1968).

If locality of reference and the presence of a page in the working set are indicators of whether a page is a good candidate for removal, it should be considered how these characteristics may be measured. To do this, the following metadata may be stored for a page:

- a *reference* flag, to indicate whether a page has been referenced (read or written);
- a *modified* flag, to indicate whether a page has been written to;
- a *timestamp* or *ordering*, to indicate when or how long ago (relatively) that the page was last referenced; and
- *counter*s, recording some characteristic of page use over time.

Using this information, various replacement algorithms have been developed. The following are a summary of these, as presented in Tanenbaum (2007):

- *Not Recently Used* — requiring reference and modified flags and often handled in hardware, the operating system will prioritise page replacement to those that do not have these flags set. The referenced flag of all pages is reset from time to time to be able to indicate whether a page has been referenced during a certain time window.
- *First In, First Out* — the operating system maintains the loaded pages as a list, from most- to least-recently loaded. When page replacement is required, the oldest is selected.
- *Second Chance* — requiring a reference bit and listing pages in the order they were loaded, the page at the head of the list is considered for replacement and given a second chance if its reference flag is set — this happens by resetting the page's reference flag and moving it to the end of the list.
- *Clock* — where Second Chance moves pages around in a list, a more efficient method is to maintain a circular list of pages with a pointer to the head of the list (clock-hand). When a new page is to be loaded, the page pointed to by the clock-hand is considered and replaced if it has been unreferenced. Otherwise, the page's reference bit is reset and the hand is moved around the 'clock-face' and the next page is considered in the same way.

- *Least Recently Used* — each time a page is accessed, it is moved to the end of the list of pages, the result being that the head of the list refers to the page that was accessed longest ago, and therefore is an excellent candidate for replacement. The overhead of updating the list at every access is often too expensive for this algorithm to be used in practice.

- *Not Frequently Used* — a poor approximation of Least Recently Used, requiring a reference bit and a page counter for each page. Like Not Recently Used, the reference flag is reset periodically, and if set, used to increment the counter. This counter gives an indication of how many times the page has been accessed.

- *Ageing* — a variant of the Not Frequently Used method, rather than incrementing the counter, the counter is first shifted right and the highest bit set if the reference flag is set. The resultant bitmask indicates the time windows in which the page was accessed, providing more information about usage patterns.

- *Working Set* — requiring referenced and modified flags, this method also requires a timestamp for each page. The referenced flag is periodically reset. When a page fault occurs, the list of pages is scanned to find a suitable page to replace. If a page's referenced flag is set, the timestamp for that page is set to the current time. If a page's referenced flag is not set and its timestamp is older than some threshold then the page is deemed to not be part of the working set and so may be replaced.

- *WSClock* — again requiring referenced and modified flags and a timestamp, this method maintains a list of pages and a clock-hand, combining ideas from the Working Set and Clock methods. Pages pointed to are considered for replacement and if not referenced, not modified, and older than the threshold are replaced. Pages that have been modified are scheduled to be written and the counter is moved on directly. Pages that have been referenced have their timestamp set to the current time and the clock hand is moved on and the next page considered for replacement.

The running time overhead of these algorithms can vary greatly depending on the amount of support provided by the system for recording the metadata.

Beyond these, a number of more advanced (and more complex) replacement algorithms have been developed that provide better performance in a range of circumstances. *LRU/k* (O'Neil et al. 1993) considers a page's k^{th} most recent access when making decisions about

which page to replace, and *2Q* (Johnson & Shasha 1994) takes this idea further by managing multiple queues to try to prioritise the replacement of pages based on frequency of use.

Working with the concept of page access recency, and maintaining page access histories even after a page has left the cache, *LIRS* (Jiang & Zhang 2002), *ARC* (Megiddo & Modha 2003), *CAR* & *CART* (Bansal & Modha 2004), and *CLOCK-Pro* (Jiang et al. 2005) provide smarter methods for managing page replacement. Where 2Q requires a selection of parameters to decide when a page should be considered to have a high access rate, CLOCK-Pro manages page classification queues dynamically. A variant of ARC is used in the PostgreSQL object-relational database system (Mustain 2005), and variants of the CLOCK-Pro algorithm is used by the virtual memory systems of the Linux and NetBSD kernels.

Additionally, cost-aware replacement algorithms have been developed for special memory storage devices, such as *CFLRU* (Park et al. 2006) and *CRAW-C* (Park et al. 2009), which take into account the different costs involved in handling writes and other operations when utilising non-disk storage.

2.1.5 Compression

In a virtual memory system with relatively slow disk storage there may be a benefit to keeping pages off-disk by compressing them in memory rather than transferring data to and from disk. The *compcache* system in the Linux kernel (Gupta 2010) is a good example of this approach. The results from compcache (2010) demonstrate clear performance benefits in a range of uses.

The benefit of page compression will depend on the processor speed, disk I/O speed, benefits of asynchronous disk I/O (if possible), size of the working set, and the size of memory used for compressed and uncompressed pages.

2.2 Caches

The difference between processor and memory speeds affects not only virtual address translation but all accesses to main memory for program code and data. Where segment and translation lookaside buffers are specialised tools to increase the performance of virtual memory use, the purpose of processor caches is to reduce the average latency for all memory accesses. This is achieved for many typical programs, but not for all. Limitations

of automatic memory caches are addressed in Subsection 2.3.1.

An automatic memory cache is a smaller, fast memory that provides access to a larger slow memory "in such a way that ... the effective access time is nearer that of the fast memory than that of the slow memory" (Wilkes 1965, p. 270). Caches are able to deliver on this performance goal due to the high degree of locality inherent in program memory access. Most general-purpose, high-performance processors feature one or more caches, as they offer large practical benefits to the performance of the machine.

For efficiency reasons, the unit of data managed within a cache is typically not a single byte or machine-word, but instead a line — a larger contiguous region of memory that will tend to be more efficient for the system to access due to memory bus design, but will also be beneficial for effective memory access time due to the spacial locality benefits of the larger amount of data (Hennessy & Patterson 2007).

If requested data is not present in a cache, the access is said to *miss* the cache. It is a *hit* if the data is present. When there is a miss, the running program will stall while the cache retrieves the data from memory (or a higher level cache).

2.2.1 Data placement

One of the fundamental choices in a cache's design is where to put data so that it may be located quickly(Hennessy & Patterson 2007).

A cache where any memory line may be placed in any cache location is described as being fully associative. Full associativity is rare in typical high-speed caches as it is often difficult to handle a cache lookup in a sufficiently timely fashion — the more places that a memory line may be stored, the more effort must be expended searching for that line. Effort in this context may be expressed in terms of time and/or hardware complexity.

The most commonly used method is to use some part of the address of the memory line as a tag to select a particular location (or set of locations) in the cache.

For a cache with n lines per set, the cache is described as being n-way set associative. For implementation efficiency reasons, n is typically a power of two. If n is one, there is only one cache line that each line in memory may be mapped to and the cache is described as being direct mapped. (As stated above, if n is equal to the number of lines in the cache, the cache is fully associative.)

Where a program requires ready access to more lines that map to the same set than will

fit in the cache, conflict misses occur — accessing lines that map to the same set will cause other, potentially desirable lines to be ejected. As such, it is clear that higher associativity results will tend to result in a lower miss rate.

Even with a fully associative cache, it is typically not possible for a cache to hold all the data a program needs. When more data is needed and there is no free space in the cache, a capacity miss occurs and some other data must be replaced (Hennessy & Patterson 2007).

2.2.2 Replacement

For caches with an associativity of $n > 1$, it is the case that there is more than one location in which a memory line may be placed within a set. A replacement algorithm is used to select an appropriate location to be replaced. Replacement in caches works under the same principle as for paged virtual memory (as covered in Subsection 2.1.4), but with a different set of constraints.

The hardware implementation of processor caches aims to maximise their performance, and require a minimum of complexity to achieve this in a cost-effective fashion. Thus implementing a minimally-complex, effective replacement algorithm for the items in a set is important.

The Least Recently Used algorithm is more feasible to implement for a very small number of slots, but many caches will instead use *Pseudo Least Recently Used* as an approximation as it is simpler, able to operate more quickly, requires less ordering state to be maintained per-set and performs comparably to Least Recently Used. Rather than requiring an ordering of all lines, the set of lines is split in half repeatedly, and one bit is used to record which half has been accessed most recently. This requires $n - 1$ bits to store the ordering of the locations in the set (IBM Corporation et al. 2009a).

2.2.3 Write policy

When performing a write, the processor must pass the relevant information to the cache, and the change must be — at some stage — propagated to main memory.

A write-through cache will pass modifications through the cache to the underlying storage straight away.

A write-back cache is one where modifications to data are stored in the cache and only transferred to main memory at a later stage — typically when the line is to be replaced in

the cache.

Write-back caches can incur extra delay when handling a subsequent miss as the program must wait for the write of a line to be issued before a read can take place to replace the line. The use of a write-back policy will typically result in less memory traffic than write-through, as a single line may be modified many times before it needs to be written back once to main memory (Goodman 1983).

Goodman (1983) also shows that the selection of a write policy relates to the problem of memory coherence in a system — that of correctly identifying the correct value for a given memory location and preventing any part of the system from accessing stale data. A write-back policy means that main memory will often not contain the most recently modified data. Even a write-through policy may not ensure that main memory contains the most up-to-date data, particularly in multi-processor systems.

For example, the PowerPC Architecture has a *weakly consistent* storage model, where

> *"the order in which the processor performs storage accesses, the order in which those accesses complete in main storage, and the order in which those accesses are viewed as occurring by another processor may all be different."* (IBM Corporation 1994, p. 333)

The benefit of such a weakly consistent storage model is in the potential for greater performance, but it does place an additional burden on the programmer and her tools to ensure a consistent processor-visible ordering through the use of explicit synchronisation operations where necessary.

For systems with multiple storage domains and separate caches for these, there is also a need to maintain coherence between caches, ensuring that consistency is maintained for shared data used by multiple program threads. A number of different cache coherence protocols have been developed for this purpose and an overview of various methods may be found in Stenström (1990), Neilforoshan (2005), and Hennessy & Patterson (2007).

2.3 Fast caches

A cache is intended to minimise the average time required to access system memory, and so it should itself operate as quickly as possible.

Hennessy & Patterson (2007) lists a number of ways that caches may be designed to provide increased performance and suggests methods to:

- structure the cache to take advantage of locality and so reduce miss rate;

- simplify the cache, to improve hit times;
- make the cache smarter, to reduce hit and miss times, and improve hit rates;
- design interfaces between the cache, processor and main store to facilitate low latency and high throughput; and
- improve compilers to reduce miss rates and miss penalties.

These methods must be carefully considered in the context of the implementation of a particular cache to offer actual benefits. Small changes to a cache design can yield large changes in overall performance (Agarwal 1989).

2.3.1 Expecting more, doing less

Caches are designed to decrease average memory access times, but there are cases where access times are increased — or not decreased as much as is desired. Efficient cache design assumes a high degree of locality present in regular program memory access patterns, but this assumption does not always hold true. Caches do not provide any guarantee on the lower-bound of access latency. When memory access patterns are sufficiently predictable to the programmer, it may be possible to populate the cache a more efficient fashion (Miller 2007).

For programs trying to maximise throughput and performance, particularly when memory access patterns are known in advance, having to wait for the cache to fetch new memory lines may be unacceptable. Greater control over what data the cache is fetching or storing may be required to provide deterministic access times for real-time applications.

For multiprocessor systems, there may be a large amount of memory-bus traffic required to maintain coherence of data between caches. It may be beneficial to be able to inform the cache that coherence is not required for particular regions to avoid.

Other aspects of cache design may cause performance problems. Memory accesses that are a multiple of cache line length will result in reduced performance with a set associative cache. To be able to change the line length, placement, or replacement algorithm may help to mitigate these effects.

A number of systems allow some degree of control by a program (or operating system) over the contents of a cache. For example, the PowerPC architecture includes instructions that provide hints to the cache regarding data that will be needed soon, as well as

indicating that blocks should be zeroed, stored, or flushed. There are also storage control attributes used to specify whether or not a location should be caching inhibited, whether coherence is required, if the data should be treated as write-through, or if the location should be guarded from speculative execution (IBM Corporation 1994). Beyond this, the Cell BE processor supports locking of particular locations into the cache (IBM Corporation et al. 2009a). Fetch hints may be used to increase performance, but they do not provide performance guarantees, where locking does.

Fetching and locking adds complexity to the design of a cache, which may not be desirable from a cost, power or scalability perspective. An alternative approach for high performance or real-time applications, like digital signal processing or real-time graphics is to use a scratchpad RAM — a small, fast, manually controlled memory. Scratchpad access times are deterministic and their contents are directly controllable by a program. They are also much simpler to implement, reducing the cost and complexity of a processor design(Banakar et al. 2002).

Processors with greater degrees of parallelism that target high performance applications and utilise specialised memory configurations include the Cell BE (Gschwind et al. 2006), Intel's Larrabee (Seiler et al. 2008) and IXP (Adiletta et al. 2002), MIT's RAW (Taylor et al. 2002), and the Tilera TILE (Wentzlaff et al. 2007).

Chapter 3

Memory and the Cell BE SPE

The memory hierarchy of the Cell BE is relatively complex and the SPE processing cores do not have the general, transparent caching design common on many other systems (IBM Corporation et al. 2009a). This thesis considers the problem of compiling existing program code to run on an SPE, minimising the changes needed to the source code of that program. In doing so, ways of abstracting the explicit memory hierarchy must be considered.

A range of other methods have been considered and developed in an attempt to simplify the use of the SPEs, to add a degree of generality and convenience and to reduce the complexity overhead required in manually managing the placement of data for these cores.

In this section, approaches to SPE memory management are reviewed and those most related to the goals of this research are identified for further consideration.

3.1 IBM SDK for Multicore Acceleration

IBM provide a range of tools for programming the Cell BE architecture in the *IBM SDK for Multicore Acceleration Version 3.1* (hereafter 'the SDK') which is available via IBM Corporation et al. (2009b). It includes a development environment, full toolchain with additional tools for processing and analysis, and various libraries. A summary of the SDK contents and use is found in IBM Corporation (2008d) which is distributed as part of the SDK.

The SDK includes a number of libraries that accelerate common high-performance computations on the Cell BE and which are optimised for (or make use of) SPEs. They include lower-level mathematical operations, linear algebra, fast Fourier transforms, Monte Carlo

methods, and cryptography. Additionally, the SDK provides frameworks to aid the development of code for various sized Cell BE system configurations. The Accelerated Library Framework (ALF) and Data Communication and Synchronization Library (DaCS) in particular are functional offload abstractions aimed at development for heterogeneous, multi-tiered systems (IBM Corporation 2008b,c).

What follows are technologies from the SDK that are of particular relevance to this research.

3.1.1 SDK software cache

One of the libraries provided is a software cache (software analogue of a hardware cache) intended to ease management of datasets larger than local store. The cache is documented in IBM Corporation et al. (2008b) provided with the SDK distribution.

This cache is able to be configured to a range of page sizes, from 16 bytes to 16 kilobytes, as well as being direct mapped or 4-way set associative. The cache uses a write-back policy.

The use of the cache requires changes to a program's source code in every location that data in main memory is accessed. Each access must be replaced with an appropriate C pre-processor macro. The cache requires a size to be chosen at compile time, and a separate cache instance is required per datatype. The cache offers explicit 'safe' and 'unsafe' access methods whereby a program may take the address of a cached object to allow faster access to it.

The SDK cache offers little type safety or compile-time assistance to aid in correct, error-free use of it by a programmer.

3.1.2 Named address spaces

Named address spaces are a extension to the C programming language standard which makes it possible to specify data as being in a particular address space and so cause the compiler to generate suitable code for handling access to the data addressed. The syntax and semantics are described in ISO/IEC TR 18037 2008.

For SPE programs this makes it possible to specify a pointer or variable with an `__ea` qualifier to indicate that the data is stored at an effective address outside of the SPE's local store. The compiler will generate appropriate DMA instruction for accesses to `__ea` qualified storage. An example is shown in Figure 3.1.

```
/* an int stored in main memory*/
__ea int i;
/* a pointer to an int stored in main memory */
__ea int* p;
```

Figure 3.1: Example of named address space syntax

Named address spaces are supported for C programs by the compilers provided with the SDK, including XLC and the Gnu Compiler Collection (GCC)[1].

To avoid performing DMA operations on every access, accesses to data outside of local store are cached in a 4-way set associative cache with a fixed, 128 byte line length. The total cache sized may be selected by an option passed to the compiler.

3.1.3 IBM single source compiler

A version of the IBM XL compiler for Multicore Acceleration for Linux is made available with the SDK. It includes "single-source compiler technology" (IBM Corporation 2008a) that permits the use of OpenMP directives (OpenMP Architecture Review Board 2008) to the compiler to generate code that utilises the PPE and SPEs (Eichenberger et al. 2006). The software cache used has two parts, a 128B line, 64KB, four-way set associative 'regular' software cache and a 32KB direct buffer used where the compiler is able to avoid cache lookup (Chen et al. 2008a). Data will be pre-fetched to the local cache, based on static analysis of the program (Chen et al. 2008b).

Liu et al. (2009) combine the analysis performed by this compiler with runtime management of DMA transfers to improve local store utilisation on the SPE.

This compiler permits existing code to be reused, but requires the programmer to mark-up code, indicating computations that are perceived to be worth parallelising across cores. This is a high-level abstraction. It removes the need for a programmer to consider separate cores, architectures or the explicit memory hierarchy of the SPEs.

3.1.4 Automatic overlay generation

For handling programs that are too large for the SPE's storage space, the SDK tools supports optional automatic overlay generation for SPE programs. The use and function of

[1]The SDK includes GCC versions 4.1.1 and 4.3.2. Support for this extension was part of the Free Software Foundation (FSF) GCC 4.5.0 release.

this feature is explained in the Programming Handbook provided with the SDK (IBM Corporation et al. 2008c).

Automatic overlay behaviour is achieved by analysing the call graph of a program and generating function stubs that will fetch and execute functions as required.

Automatic overlays are generated for program text only.

3.2 COMIC

Lee et al. (2008) present a coherent shared virtual memory system for the SPE that utilises a page buffer in the local store of each SPE. The cache is described as using a lookup mechanism similar to a hardware 4-way set associative cache, with separate read and write tag indexes and an index size larger than the number of available pages to reduce conflict misses.

COMIC is a robust abstraction that provides a coherent, shared-memory, parallel processing environment, but which requires a large degree of manual modification to a program to make use of it. SPE functions are scheduled and executed from a PPE thread, and explicit accessor functions must be utilised in a program to access shared storage. While it appears possible that with some effort much of the required runtime program infrastructure could be generated automatically, existing programs do require substantial changes to make use of this system.

3.3 "A Novel Asynchronous Software Cache"

Balart et al. (2008) presents a fully associative software cache implemented as a runtime library that works in concert with a specialised compiler. Many of the cache parameters are selected at compile time by the compiler after analysing the code being compiled. Particularly, the length of cache lines will be selected to minimise conflict misses.

The compiler will associate variables with particular cache locations and will insert extra loop levels to allow traversal of data ranges without needing to check cache presence on every iteration. Benefits of this approach are that variables are directly associated with cache locations in a 'look up and translating table' and so may be located quickly, and the compiler may add pre-fetches and issue write-back operations when data is no longer required to help reduced the overhead of DMA operations.

3.4 "Hybrid Access-Specific Software Cache Techniques"

Gonzàlez et al. (2008) describes a hybrid cache, with separate cache mechanisms for memory locations with high and low spacial locality, as determined by the compiler. High locality accesses are expected to occur with a high hit rate, and so the high locality cache is optimised for quick lookup. The low locality ('transactional') cache is expected to provide a relatively low hit rate and is optimised for handling misses with as little latency as possible. The high locality cache has 64KB of storage, with lines from 512B to 4KB. The low locality cache has a capacity of 4KB, made up of 32 128B lines. Both caches are fully associative.

3.5 Codeplay Offload

Codeplay Offload is a framework designed to aid programmers to make use of the SPEs for computation (Codeplay Software Ltd. 2010b). Offload will generate code for SPEs based upon appropriately annotated C++ code, not dissimilar to IBM's XLC.

The generated code will make use of a software cache as needed. The software cache implemented is four 128 byte lines, with direct mapping, which allows for fast lookup as only one comparison is required to detect if a line is present (cache lookup is claimed as taking 18 cycles).

This cache performs fine-grained tracking of changes to data. A modification mask is maintained for each line, recording for each byte whether it has been modified or not. Lines are written back using an atomic get-merge-write process (Codeplay Software Ltd. 2010a).

Additionally, the software cache holds pointers to cache lookup indexes in assigned registers at all times, which contributes to the fast lookup time.

3.6 Other

There are a range of other libraries and languages that make use of the SPEs for computation. These include:

- Cellgen(Schneider et al. 2009) is another tool that uses OpenMP-like annotation of program code to generate programs utilising the PPE and SPE. It uses programmer-provided annotation of data accesses to avoid the need for a runtime software mem-

ory cache, performing statically scheduled memory transfers as required but supports only a limited number of program forms.

- RapidMind (McCool 2006) is a dynamic runtime compilation system for the Cell BE, GPUs, and Intel/AMD processors, which is implemented as a compile- and run-time library. RapidMind requires the use of domain-specific C++, which does provide cross-platform compatibility (on supported platforms), but specialised to the RapidMind system.

- Sequoia (Fatahalian et al. 2006) is a programming language that emphasises data-expressive forms and explicit data movement, using the parallel memory hierarchy programming model. It is inherently a data-oriented language, and has support for the Cell BE processor. Knight et al. (2007) detail the implementation of a Cell compiler for Sequoia. Houston et al. (2008) present a more generalised runtime abstraction for Sequoia.

- Vasudevan & Edwards (2009) demonstrate a compiler for the SHIM programming language targeting the Cell BE, using the SPEs for computational acceleration, with DMA operations and mailboxes for messages and synchronisation.

- CellSs (Bellens et al. 2006) is the Cell Superscalar framework, which is a source to source compiler that works with OpenMP-like program annotations.

- Ferrer et al. (2008) presents an evaluation of the SARC programming model for the Cell BE that makes use of the CellSs interface.

- Charm++ Kunzman & Kalé (2009) is an acceleration method that is similar to CellSs.

- Kudlur & Mahlke (2008) and Choi et al. (2009) each present a statically partitioned data model for stream processing on the Cell BE.

The use of program annotations, programming languages apart from C/C++, and the use of custom libraries takes these further away from the goals of the hoard, particularly for the reuse of existing program code. Additionally, these methods do not address the problem of efficient caching or virtual memory for the SPE in any great depth.

Other work related to software caching includes:

- Ahmed et al. (2008) presents a SPE nano kernel system (SPENK) that allows scheduling of multiple micro-threads on each SPE. Using the SPE's interrupt mechanism to

trigger context switches, SPENK helps to reduce DMA multi-buffering and scheduling complexity in programs.

- A compiler-generated data caching framework, FlexCache is described in Moritz et al. (2001).
- Flexicache, a specialised software instruction cache for the MIT RAW architecture, is presented in Miller (2007).
- Werth et al. (2009) presents a code cache for the Cell BE SPE that aims to reduce the amount of program code that is contained in an SPE's local store.
- For GPUs, Baskaran et al. (2008) describes an automatic data management system for multi-level parallel architectures with explicit hierarchy systems while Silberstein et al. (2008) gives consideration to some of the difficulties involved in implementing a more general read-only software cache for a GPU.

These again address different problems to the hoard, although are more complementary to the hoard's goals, addressing problems that the hoard is not able to solve, such as program size and methods to generate more efficient programs.

3.7 Summary

Of the frameworks, methods, and techniques listed above, many are dependent upon the use of a particular programming language or program annotations and compiler analysis to generate or inform an automatic memory manager for SPE programs. A primary goal for this research has been to minimise the amount of explicit analysis or modification a programmer must make to a program in porting it to the SPE, while offering reasonable performance, with the flexibility needed to be able to optimise a program as appropriate for the architecture.

For this reason, the methods that match most directly are COMIC, named address spaces and the IBM SDK cache.

COMIC has an optimised cache design, offering decent performance. It does requires substantial modifications to existing programs, and requires a PPE control thread for the running program. COMIC is designed as a shared virtual memory system, where this thesis considers only a standalone method. Due to its greater complexity and scope, and

due to source code for COMIC becoming available only late in this research, COMIC will not be used for a direct comparison. A deeper analysis of COMIC and comparison between it and the performance characteristics of the hoard is included in Section 8.9.

Codeplay Offload also appears to be a interesting candidate for further comparison with techniques from the hoard and COMIC. Its release occurred late in this research and there has been insufficient time to give it full consideration.

The named address space implementation for the SPE in GCC uses a cache that is very similar to that of the SDK cache with a four-way set-associative lookup. It lacks most of the configuration options of the SDK cache — only its total size may be directly controlled. The compiler generates code in a very similar fashion for accesses to `__ea`-qualified locations to that which is generated when using the SDK cache. Some initial tests using the `__ea` address space indicated that performance was comparable to a similarly configured SDK cache.

Initial testing also revealed that there were a number of bugs in the named address space implementation of the SDK compilers and a pre-release version of GCC-4.5. These bugs prevented the compilation or correct function of the benchmark programs used in this research. The final release of GCC-4.5 had a greatly improved implementation of the extension, but by then there was insufficient time available to attempt to re-visit its use. For this reason, combined with the limited configurability and observed similarities with the SDK cache, named address spaces will not be used for comparison on this thesis.

The SDK cache requires explicit substitution of memory accesses with macros. This is intrusive and error prone, but requires no explicit support for caching from the compiler. The SDK cache is also implemented in the form of a traditional four way set associative hardware cache. *This cache will be used as the point of comparison for the memory abstraction developed in this thesis.*

Characteristics of other methods will be considered and tested, where applicable. In the next chapter, the architecture of the Cell BE will be examined in more depth.

Chapter 4

The Architecture of the Cell Broadband Engine

The hoard has been developed to facilitate reuse of existing program code on the Cell Broadband Engine processor, and to do so in a way that makes use of the features and design of this processor in the best way possible. This chapter describes key features of the hardware which are relevant to the implementation of the hoard. A more thorough description of the hardware may be found in IBM Corporation et al. (2009a).

4.1 Background to the processor

The Cell Broadband Engine Architecture (CBEA) was developed through a joint venture of Sony, Toshiba, and IBM. The architecture is implemented in two processor models, the Cell Broadband Engine (Cell BE) processor and the PowerXCell 8i, a refined model that offers increased double-precision floating point performance and support for different memory technologies.

The Cell BE processor was initially designed to be used in the Sony PlayStation 3 and other media-rich consumer-electronics devices, and it was expected that the architecture would also support a broad range of commercial and scientific applications (IBM Corporation et al. 2009a).

Because the processor was being designed to be used in a high-end game console, performance was one of the most important goals, set against a tight limit on power consump-

tion and heat generation (Shippy & Phipps 2009, Hofstee 2005a).

IBM Corporation et al. (2009a) and Hofstee (2005b) sum up the constraints as three "Limiters to Processor Performance" (Hofstee 2005b, p. 4) — the power wall, the memory wall, and the frequency wall.

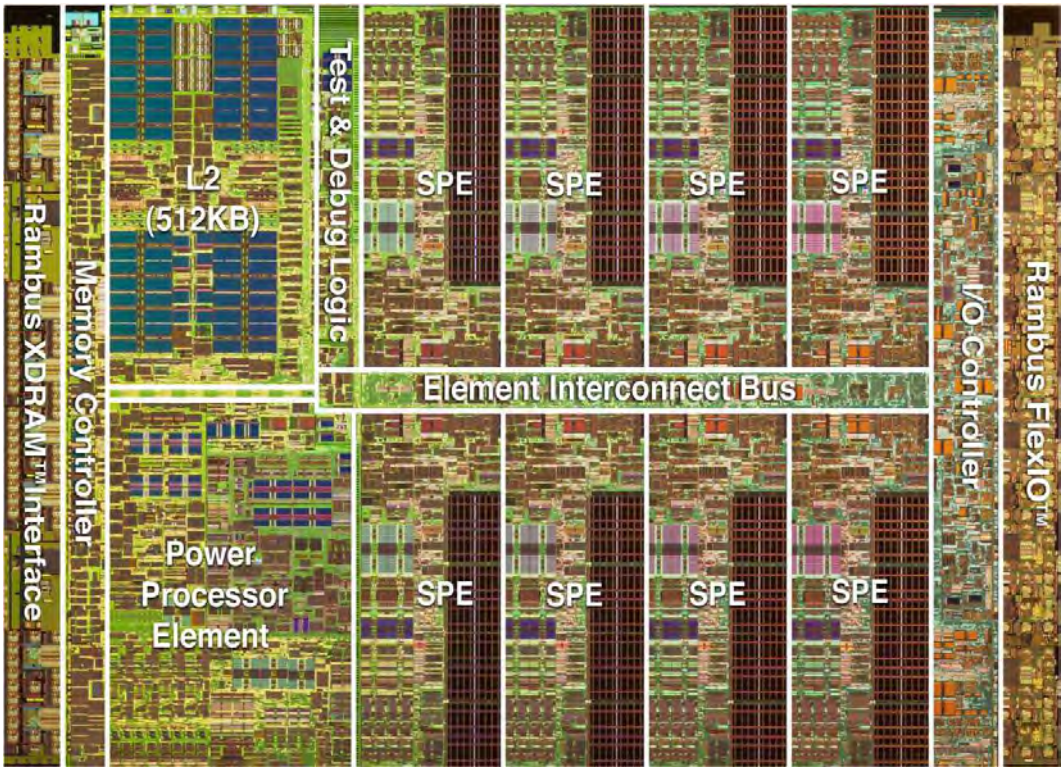


Figure 4.1: Layout of the Cell Broadband Engine processor. Source: IBM Corp.

4.1.1 Power wall

The relationship between power consumption and performance is a critical detail for the use of processors in many contexts. The Cell BE design seeks to maximise performance per watt by maximising computing power and avoiding the use of silicon 'real-estate' that does not pay for itself in terms of performance (Hofstee 2005a).

In the Cell BE, a large amount of space is given over to computation cores (the SPEs), where most of the computing power of the processor is available — more than 25 GFLOPS per SPE at a clock speed of 3.2GHz. The SPEs are designed to maximise computation throughput. They work most efficiently for statically-scheduled program code, and large-scale data processing. In contrast, the single PPE is designed to handle the operating system and other control-intensive code (IBM Corporation et al. 2009a).

Because of this design approach where different types of cores each serve particular purposes, the processor is able to deliver very high performance per watt (Hofstee 2005b, Williams et al. 2006, 2008).

There is a practical cost associated with this design as more work falls to the programmer and/or compiler to generate programs that make full use of the processor, utilising its various resources effectively.

4.1.2 Memory wall

Wulf & McKee (1995) observed that, at that time, memory speeds had been increasing at approximately 7% per year and CPU speeds at 80%, and point out that at some point it will not be possible to supply data to a fast processor fast enough to keep it fully utilised — they identify this as the memory wall (Figure 2.1 illustrates this divergence).

As was shown in Subsection 2.1.3, the time it takes to provide data to the processor from main memory can be many hundreds of clock cycles — time when the processor may be left unable to do any work. Larger caches do help to reduce the average memory access time, and can help to reduce overall memory traffic as every cache hit results in one less transfer across the system's memory bus.

In many multi-processor or multi-core systems, the amount of bus traffic needed to keep transparent, automatic caches coherent can quickly increase to a level where other performance suffers.

To combat this, the Cell BE SPEs feature a three-level memory hierarchy — a large register file, fast locally addressable memory in place of cache, and the system's main memory. The program running on each SPE must manage the use of its local store memory, and is able to instruct the SPE's memory flow controller (MFC) to transfer various sized blocks of data to and from local store. The SPE's local store has much in common with scratchpad RAM, described in 2.3.1.

Williams et al. (2008) has an example of the problem of high bus traffic influencing performance in the optimisation of a lattice Boltzmann simulation. The per-core performance the tested Itanium2 and Clovertown systems scales poorly as the number of in-use cores increases. In contrast, the Cell BE maintains a consistent per-core performance independent of the number of cores in use.

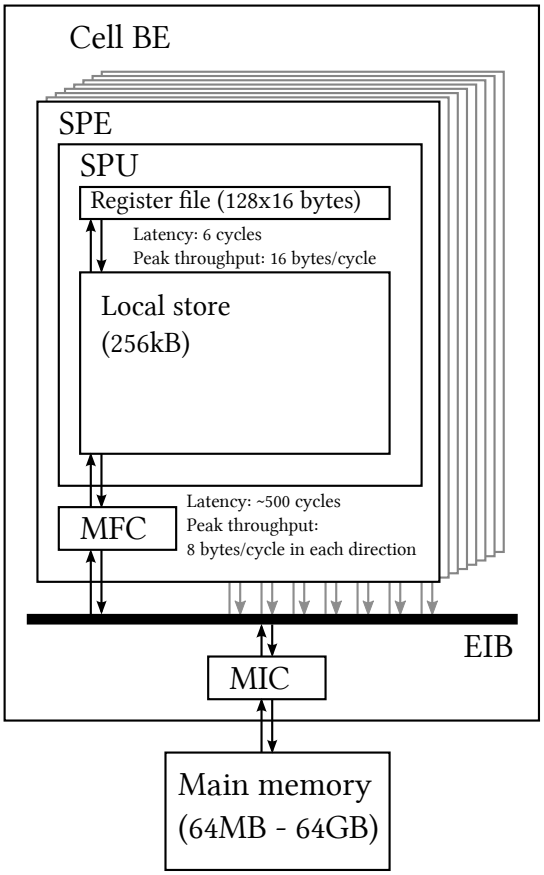


Figure 4.2: The Cell BE memory hierarchy, latencies and sizes.

A large register file makes it possible to keep the SPE at a high level of utilisation, in many cases hiding the high latency between the SPE's local store and main memory (Hofstee 2005a). The explicit management of local store means that the programmer has full control of all memory bus traffic from the SPEs. The MFC accesses system memory without interrupting the program running on the SPE, and up to sixteen transfers may be in flight simultaneously (IBM Corporation et al. 2009a).

4.1.3 Frequency wall

High frequency is one key way to achieve higher performance, although there is a trade-off between frequency and efficiency. Higher frequencies lead to greater power consumption and thus heat generation.

Greater complexity in a processor can also reduce the maximum operating frequency for the processor. In addition, greater complexity does not automatically lead to greater performance. The design of the Cell BE seeks to balance complexity with performance,

leaving out features that do not provide sufficient performance gain for their complexity.

For example, rather than adding more homogeneous cores, the Cell BE balances the general purpose control of the PPE with the small size and high computational power of the SPEs. The SPEs do not have common features like out-of-order execution, dynamic branch prediction, or an automatic level 1 cache but rather rely on the static scheduling of instructions, static branch prediction and the programmer to manage the use of local store memory.

Operating frequency may also be increased by adding extra pipeline stages for instruction execution. Longer pipelines do increase difficulties in scheduling instructions for execution due to data dependencies, but the SPE's large register file helps to combat this (Hofstee 2005b).

4.2 Processor features

Figure 4.3 is a simplified structural diagram of the Cell BE processor, with its various processing cores and interfaces to the broader system. The design and architecture of the Cell BE are presented in great detail in IBM Corporation et al. (2009a). Material of particular relevance to this thesis are summarised in this section.

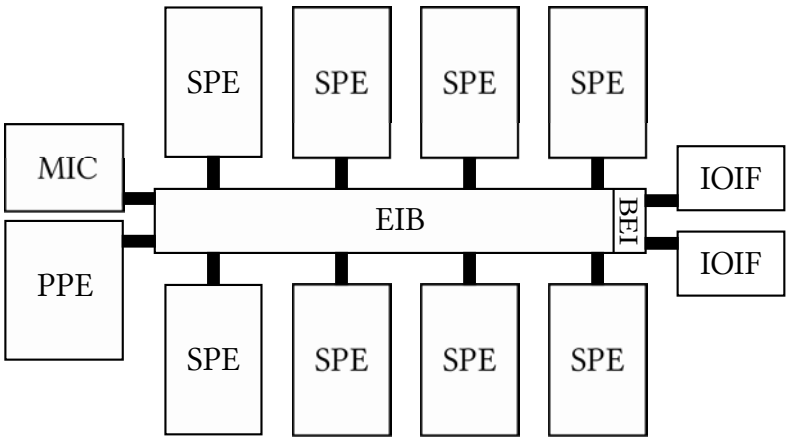


Figure 4.3: High-level structure of the Cell BE processor

4.2.1 The Power Processor Element

The primary processing core of the Cell BE is the referred to as the Power Processor Element (PPE) and conforms to the PowerPC architecture version 2.02, with the Vector/SIMD Multimedia Extension (VMX), and is compatible with the PowerPC 970 processor core (IBM Corporation et al. 2008c). The system's operating system runs on this processor and it is described as being optimised for control-oriented programs.

The PPE supports two simultaneous threads of execution, has separate 32 KB instruction and data caches and a 512KB combined level 2 cache.

The PPE and SPEs are connected to the Element Interconnect Bus (EIB), a four-lane ring bus that provides communication between the processing cores and system memory. Each lane is 16 bytes wide, and the EIB has internal maximum bandwidth of 96 bytes per cycle in 12 simultaneous transfers (Hofstee 2005a).

The PPE supports 32 and 64 bit address space modes of operation.

4.2.2 Synergistic Processor Elements

The Cell BE processor contains eight Synergistic Processor Elements (SPEs) — processing cores optimised for data-intensive computation (Gschwind et al. 2006). The SPEs are designed with an instruction set focused on SIMD computation.

Each SPE has a locally addressable memory of 256KB, known as local store, and there is no automatic memory cache as part of the SPE. The local store is used for both program code and data. SPE programs are loaded into an SPE's local store by the PPE, and the SPE is able to control the arrangement of program code and data in local store through the use of the memory flow controller (MFC). The MFC provides a direct memory access (DMA) engine that may have up to sixteen individual transfers in flight at one time, moving data to and from main memory or the local storage of other SPEs. Each SPE is capable of up to 25.6GB/s in and out (for a total 51.6GB/s) via its MFC (Kistler et al. 2006).

DMA operations may be reordered by the MFC to maximise throughput. Each DMA operation may be assigned a tag ID, which is then able to be used to check or wait for the completion of that operation. The MFC permits non-blocking checks of progress as well as blocking calls that return when all outstanding transfers within a group are completed. Execution of a program on the SPE is blocked if the issue of a DMA operation is attempted while the queue is full, with sixteen operations pending completion.

If an explicit ordering of DMA operations is required, reordering may be explicitly limited or prevented by specifying that the operation have a fence or barrier. All DMA operations are issued with a particular tag ID, from 0 to 31. A fence ensures that a particular DMA operation will occur only after other DMA operations with the same tag ID. A barrier prevents ensures that the particular operation will not be ordered before any preceding, or after any later-issued DMA operation.

The MFC also supports DMA list operations, transferring data between a range of variable sized regions and a contiguous block of local store memory using a list of addresses and sizes to control the transfers. Scatter-gather style transfers may be implemented through the use of DMA lists (IBM Corporation et al. 2009a).

MFC transfers are performed on multiples of memory lines, each being 128 bytes. Smaller transfers have the same latency as whole-line transfers. MFC transfers of data not aligned to a 128 byte boundary will also transfer the entirety of any partial line. Transfers that have 128 byte-aligned source and destination addresses, and that are multiples of 128 bytes are the most efficient.

Each SPE has 128 general purpose registers, each 128 bits (16 bytes) wide. Data transfers between local store and registers may only be performed 16 bytes at a time, from an aligned address. Unaligned accesses require additional manipulation.

The SPEs local store is single-ported, meaning that only one access at a time is permitted. This means that DMA operations, load and store instructions, and instruction pre-fetch for execution will have some — typically small — effect on one another.

The SPE's registers are general purpose — able to hold the operand or result of any instruction. Most instructions will treat source registers as vectors of smaller scalar values — sixteen bytes; eight halfwords; four words; four single-precision or two double-precision floating point items; or as a single array of 128 bits (IBM Corporation et al. 2007). A whole sixteen byte register is referred to as a quadword.

All integer and floating point computation instructions will act in a SIMD fashion on the contents of a register. The SPE is able to issue a four-element single-precision floating point fused multiply-add instruction every clock cycle, providing a peak performance of 25.6GFLOPS for each SPE. Double-precision floating point performance is much poorer on the Cell BE where each double-precision operation stalls all instruction execution for six cycles. The PowerXCell 8i processor eliminates this stall (IBM Corporation et al. 2009a).

While both single- and double-precision floating point representations used match IEEE Standard 754, single-precision calculations are not fully compliant with that standard. Details may be found in IBM Corporation et al. (2009a).

For instructions that use only one scalar operand, or produce only a scalar result (e.g. branches, loads, stores, constant creation, etc.), the vector's 'preferred slot' — defined as illustrated in Figure 4.4 — is used. If data for some operation are not located in the preferred slot, a shift, rotate or shuffle instruction will be required to move them.

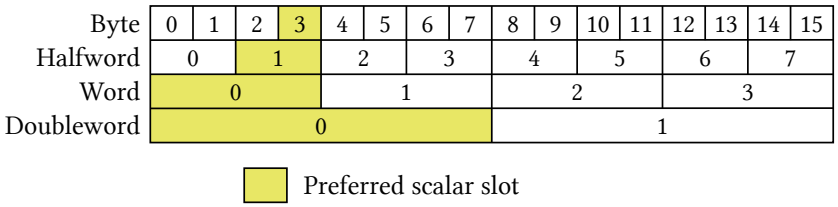


Figure 4.4: SPE vector register usage and preferred slots

The application binary interface (ABI) for the SPE is defined in such a way that all scalar variables kept on the stack are stored in the preferred slot of an aligned quadword. This ensures fast access to these variables, at the cost of the use of some extra space (IBM Corporation et al. 2008e).

Memory accesses consider only the lower eighteen bits of any address used (that is, the 256KB region). This means that any attempt to use an address with higher bits set will be performed modulo 256K (2^{18}). There is no hardware checking of addresses before they are used and no exceptions or interrupts for accesses to 'invalid' memory locations. Load and store instructions will zero the lower four bits of any address, loading the aligned quadword containing the requested location.

For access to system memory, each SPE has a 256 page entry TLB. TLB misses can be handled by hardware or software, depending upon system configuration.

Each SPE also has an eight entry SLB. Each memory segment on the system is 256MB, and each segment miss is handled in software, by the operating system. Kerr (2009) demonstrated that random access to a memory region larger than 2GB (eight segments) results in very large performance penalties for SPEs accessing system memory access.

4.2.2.1 Size and alignment

As mentioned previously, load and store operations act on aligned 16 byte regions in local store memory, and DMA operations are most efficient when accessing aligned 128 byte regions. There are, however, further restrictions on size and alignment for DMA:

- transfers can be from 0 to 16,384 bytes in size;
- transfers of 16 bytes or more must be multiples of 16 bytes, with 16 byte aligned source and destination addresses; and
- transfers smaller than 16 bytes may be 1, 2, 4 or 8 bytes large, the source and destination must be naturally aligned, and must share the same offset from a 16 byte boundary.

SPE instructions are 4 bytes long and 32 instructions at a time are loaded from local store (128 bytes), and are held in an execution buffer. There are two execution buffers.

The SPE has instruction execution pipelines, described as odd and even. Each pipeline will execute only certain instructions. For example, loads and stores are executed on the odd pipeline and floating point operations are executed on the even pipeline. Instructions are able to be issued simultaneously (dual-issued) if one does not depend on the other, and if they are located at particular local store locations — even pipeline instructions at addresses ending with `0x0` or `0x8` ($address \bmod 8 = 0$), and odd pipeline instructions at `0x4` and `0xC` ($address \bmod 8 = 4$). Instructions not located at a preferred offset cannot be dual-issued (IBM Corporation et al. 2009a).

4.2.2.2 The place of the SPE in the chain of command

> ***Jayne Cobb**: You know what the chain of command is? It's the chain I go get and beat you with 'til ya understand who's in... command here.*
>
> *(Firefly Episode 1: The Train Job 2003)*

> ***Bender:** And what's Peter Parrot's first rule of captaining?*
>
> ***Fry:** Always respect the chain o' command ...*
>
> *(Futurama Episode: The Birdbot of Ice-Catraz 2003)*

Programs loaded to the SPEs run in their own local memory space. For an SPE program to alter the wider system state, the program must transfer data, send messages, or send signals.

The primary programming model supported through the IBM Cell SDK for Linux is that each SPE program context is managed by a separate POSIX thread running on the PPE. Mostly, these PPE threads need no active processor time while the SPE program runs, but it is through the PPE thread that the program on an SPE is able to interact with the operating system or other applications in the system (IBM Corporation 2007).

The software provided to allow the SPE to interact with the wider system consists of a C runtime library on the SPE (a port of Newlib (2010) to the SPE), and the libspe2 library (IBM Corporation et al. 2008d), used by a PPE program that wishes to utilise SPE contexts. These two libraries provide the necessary mechanisms to allow an SPE program to make system calls and to manipulate resources outside of the SPE.

When an SPE program makes a system call or other library function call, it accesses a stub function that saves the arguments for the function into local store, stops execution of the SPE, and signals the PPE thread with an identifier to indicate the system call to be executed and the address of the arguments in the SPEs local store.

The PPE thread transfers the arguments from the SPE's local store, performs the desired operation, pushes the result back to the SPE, and restarts its execution.

It should be clear that there is a large overhead to making system calls, library or other PPE calls — many thousands of cycles are spent signalling, performing DMA operations, and waiting for the results of the operation.

For message passing between PPE and SPE threads, the Cell BE supports a number of mechanisms, including various types of signalling, 32 bit mailboxes, DMA, and atomic operations. These are typically asynchronous — leaving the SPE able to continue executing while messages are delivered — and are much lighter-weight than the PPE callback mechanism detailed above (IBM Corporation et al. 2009a).

4.2.2.3 Operating environment

SPEs do not run an operating system and have no supervisor mode. There is no direct support for multi-threading on a single SPE, although some frameworks add a cooperative form of this (e.g. SPENK, mentioned in Section 3.6). When running Linux, there is support

for context switching programs running on an SPE, but this comes with a large overhead as all of the SPE state must be saved and restored, taking thousands of cycles. (The MARS system allows a lighter-weight context switching to take place, through the use of small manager kernels (Levand 2008).)

There is no operating system level dynamic linker, and so SPE programs do not inherently support any kind of dynamic runtime linking. Because the SPE runs without memory protection, and on the SPE "code is data" (Acton 2007, p. 3), it is possible for a program to dynamically load and modify the program text present in local store. Levand (2008) and Acton (2007) demonstrate this kind behaviour.

Additionally, there is no support for C++ exceptions on the SPE in GCC or XLC (IBM Corporation et al. 2008a).

4.3 Access to the Cell BE

Apart from the PlayStation 3, the Cell BE and PowerXCell 8i have been utilised in a range of products. These include IBM BladeCenter modules, products from Mercury Computer Systems, including HPC and specialised commercial products, as well as products from Fixstars Corporation. The Roadrunner supercomputer at Los Alamos National Laboratory utilises PowerXCell 8i processors, and featured as the fastest in the Top500 lists from June 2008 to June 2009 (Top 500 Supercomputer Sites 2010).

The Cell BE processor pictured in Figure 4.1 shows eight SPEs, but the processors used in PlayStation 3 consoles have one less hardware SPE available — a change made to increase the effective production yield, allowing the use of Cell BE processors containing up to a single faulty SPE. One other SPE is reserved for the use of the console's hypervisor, leaving six available for general purpose use (Kurzak et al. 2008).

At launch, the PlayStation 3 allowed the installation of another operating system (an *OtherOS*) apart from the default OS used to launch and manage games and other entertainment content (the *GameOS*). In 2009, a new revision of the PlayStation 3 hardware was released by Sony, and the support for the OtherOS feature was removed from the newer, slimmer model. Further, Sony released an 'optional' system software update for the PlayStation 3 on 1 April 2010 in which the OtherOS functionality of the console was removed.

The primary "other" operating system available for use on the PlayStation 3 is Linux, launched via an updateable bootloader, stored in the system's firmware. A range of Linux distributions are available for the PlayStation 3, offering similar features and allowing full utilisation of all available processing cores on the Cell BE's, PPE, and SPE.

PlayStation 3 consoles have a fixed 256MB of system RAM, an internal SATA HDD, USB, gigabit and wireless networking, and have found a range of non-gaming uses including the PS3 Gravity Grid at the University of Massachusetts, Dartmouth (Khanna 2010), password brute-forcing by the U.S. Immigration and Customs Enforcement Cyber Crimes Center (Szydlowski 2009), protein folding through the Folding@home project (Folding@home 2009), and are also used by the U.S. Air Force for image processing and other applications (Hoover 2009).

The research in this thesis was performed primarily using a PlayStation 3 console running Debian/Sid Linux (Debian 2010). Programs have been cross-compiled using FSF GCC 4.4.1 (Free Software Foundation 2010b), with tools and libraries from the IBM SDK for Multicore Acceleration version 3.1 (IBM Corporation et al. 2009b).

Some early investigation was done with access to a QS20 blade server, generously made available by the Barcelona Supercomputing Center.

Chapter 5

Architecture of the Hoard

5.1 Origins and function

This research began with an exploration of the complexities and overheads of accessing memory from the Cell BE SPEs. Some very simple programs were written and some unsurprising results obtained:

1. Naïve access to memory locations outside the SPE is very slow.

2. Even static memory access patterns can be difficult to schedule in an efficient fashion.

The first point is not surprising. The Cell BE is designed in a way that ensures high-speed transfer of large, aligned, contiguous blocks of data, as is detailed in the previous chapter. The time cost of transferring small, disconnected, unaligned pieces of data is very high — as shown by the performance of small transfers as presented in Gschwind et al. (2006). The first point is true for many high-speed architectures that have a large latency for accesses to main memory. Without consideration of the interaction between data layout and the structure of the memory hierarchy, performance will suffer greatly. An example of this is presented in Albrecht (2009).

With regard to the second point, the experience of trying to tune a very simple program for the SPE's MFC made it clear that it is not a simple task to explicitly handle memory accesses outside the SPE's local store. The task of managing memory becomes an additional and unfamiliar problem for the programmer to address. The additional handling of memory accesses can obscure program flow and logic. A system detail that is invisible to

programmers on other architectures becomes a complex, error-prone addition to the process of writing a program. Denning (2005) estimates that manual memory management in such a system at least halves a programmer's productivity.

These 'discoveries' are nothing new. They are problems that have existed for as long as different-speed memories have, and are problems that can, perhaps, be reasonably considered to be solved through the use of virtual memory and automatic caching (Denning 1970).

But programs running on SPEs do not directly have access to these features, and this is by design. The presence (and lack) of features all relate to the goal of producing a processor that meets particular performance, cost and power constraints (Hennessy & Patterson 2007, Hofstee 2005a), as was discussed in Section 4.1.

Due to the complexity of the architecture and the difficulty involved in reusing existing source code for SPE programs, it was decided to investigate methods of simplifying the effort of programming for this platform. Throughout the design process, the focus has been on porting existing programs to the hoard, with no up-front focus on particular program types. In this sense it can be considered that the hoard is designed to be "general-purpose" and suitable for use with a range of programs. In practice, SPEs have a design that clearly suits some types of problems more than others — where possible, it has been the intention to use strengths of the SPE design to create a tool with broad usefulness.

The first direction in approaching this problem was to consider how an automatic software cache might be implemented. As mentioned above, a cache adds overheads in chip real estate and speed when implemented in hardware, and overheads would seem to be no less a problem when written as software. (Miller 2007 quantifies overheads in software and hardware caching.)

For cached data, there will be a time penalty introduced for all accesses — memory accesses will need to be checked, even if the data is present in local store. The software cache will add complexity, increase power consumption and will typically result in an increase in memory traffic between local store and main memory when compared with an 'optimal' memory access pattern.

All these things are true for any memory hierarchy — 'automatic' caching will be slower than an optimal, programmed use of a fast, local memory. But is an optimal, programmed solution actually possible? Is a 'perfect' solution? Does it even matter?

Software development is expensive. An investment in faster hardware will often provide a solution more cheaply and easily than paying expensive programmers to squeeze every last cycle out of a particular system. (Atwood (2008) makes a strong case for investing in hardware before directing expensive programmer resources to the task of optimisation.)

Code re-use is one way to reduce development costs — taking existing, functional code and re-using it in a new context. But for the Cell BE it is not a simple task to recompile existing programs and gain the full benefit of the processing power of this processor. The explicit memory hierarchy means that a great deal of additional work must go into making the interactions between the SPE and main memory correct and efficient.

Because of its flexibility, a software cache has particular advantages over a hardware cache — it may be tuned or modified to suit the specific needs of a program, or removed altogether. A software cache gives the greatest level of flexibility. It allows for simplicity and convenience in general, with reasonable performance while allowing specific, high performance implementations where needed.

This thesis demonstrates that, rather than a software cache, a paged memory system can simplify the process of porting existing programs to the Cell BE SPE, and will simplify and improve the performance of a range of programs.

5.2 The evolution of a design — it starts with a name

As mentioned in Chapter 1, this research began with the idea of a software cache, but the name hoard was suggested[1] and the structure of this system has developed into something with little resemblance to a traditional hardware cache.

This remainder of this chapter documents the development of the hoard from its exploratory beginnings to a point at which it is deemed to be sufficiently functional and the complexities of the design of the SPE are well enough understood. The ideas and motivations that have gone into the design of the hoard are also covered in this chapter.

This chapter describes the development of the hoard from the initial ideas, through a number of revisions of structure and function to arrive at a completed design. The chapter records some ideas and decisions that were tested and removed from the hoard during its development.

[1] There is a Hoard memory allocator project available via http://www.hoard.org/. It seems unlikely that there will be any confusion between that work and this research.

In Chapter 8, the final hoard design from this chapter is tested and evaluated in detail. Based on those observations, Chapter 9 details further refinement and testing of the hoard.

5.3 A first cut

The first prototype consisted of ideas pulled somewhat out of the air, and the implementation was tested and iteratively refined from there.

Knowing that larger DMA transfers are more efficient than smaller ones (Gschwind et al. 2006), the first page size chosen was 1024 bytes — a number larger than the memory line size of 128 bytes and small enough that 'many' of them would fit with a program in the SPU's local store.

The first implementation was added as a modification to an existing program, with managed accesses to an array integrated into a smart-pointer style C++ class (not dissimilar to the ideas presented in Horstmann (2000)). As mentioned in Chapter 4, the SPU does not perform any check that addresses point to a valid location in local store — all addresses are considered to be valid, and accesses to local store are performed modulo 256K (2^{18}). In addition, there is no memory protection — a program may access any local store location.

In the smart-pointer style class implemented (called `paged_array`), data access was handled through an overloaded index operator (`operator[]()`) that, when called, would check an index table to see if the required page was loaded, fetching it if necessary and returning a reference to the appropriate location. In this way, a `paged_array` object may be treated as if it is a regular array, hiding the detail of page lookup from the running program.

A write-back policy was chosen as it seemed beneficial to perform few DMA operations. For the sake of simplicity, page replacement was handled using a FIFO method — no usage or modification tracking of pages was included, so all pages were written back to system memory after use.

5.4 The Descriptor

> *"Descriptor — A computer word used specifically to define characteristics of a program element. For example, descriptors are used for describing a data record, a segment of a program, or an input-output operation." (Burroughs Corporation. Sales Technical Services 1961, p. C-1)*

The first and fundamental part of implementing the hoard came from some observations about the design of the SPE instruction set and local store memory infrastructure:

- Loads and stores (between local store and registers) are all 16 bytes large and 16 byte aligned.
- Scalar operations act upon the preferred slot of a 16 byte vector — in the case of a vector of words (int, addresses, etc), the first four bytes.

A corollary of these points is that random, unaligned access to scalar data will be slower than access to data that is stored in an appropriately aligned memory location. For unaligned data, the offset must be calculated and a transformation of a loaded vector performed. Data not stored, accessed or loaded via the preferred slot comes with a time penalty.

In practice, the difference is not typically more than two instructions. It may seem that two instructions is not a big penalty, but for memory accesses that occur tens of millions of times every second, small changes can yield large performance increases.

For any type of cache, to read data from the cache requires that the system:

- check that the data for the requested address is loaded into the cache; and
- load the actual data from the cache to a register.

This is done for every memory access, hit or miss, so it should be as fast as possible.

If the local store address (lsa) is stored in the preferred slot of a register (as was shown in Figure 4.4, and specifically in Figure 5.1), which is stored aligned in local memory, it is then possible to quickly:

- load the appropriate vector from local store to a register;
- check to see if the address is set, indicating that the page requested is also loaded; and
- use it as an address and to load the desired page.

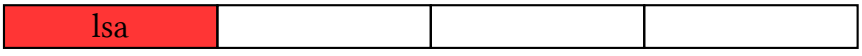
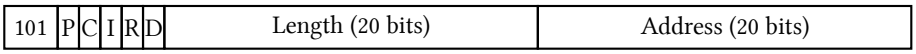


Figure 5.1: Local address in a quadword

So far, there is a single local store address (four bytes) stored in a sixteen byte region of local store. The unused space may be used to store associated data — items that are relevant and related to the local store entry, but used less frequently.

Managing memory regions in this way, with a collection of data items stored together in a single machine word, has a lot in common with the data descriptors that featured in the Burroughs B5000 (and subsequent Burroughs large systems) in the early 1960s.

The Burroughs systems used a 48 bit machine word to store an address, a length, and some information bits to describe a memory allocation, pictured in Figure 5.2. The presence bit was used to indicate whether the address was for a location in core memory or if it had been paged out to disk. If the address itself was zero, space for the data had not yet been allocated.



P Presence
C Copy
I Indexed
R Read-Only
D Double Precision

Figure 5.2: Burroughs B6700 data descriptor format, from Hauck & Dent 1968

Through the use of descriptors, the Burroughs systems implemented a number of useful features including:

- paging of virtual memory;
- just-in-time (JIT) memory allocation;
- bounds checking on arrays and allocations;
- recording of a data-type for an array; and
- providing access control to program code and data.

This list of features covers much more than typical memory management. More information on the Burroughs data descriptor format may be found in Hauck & Dent (1968).

Ideas from the Burroughs data descriptor influenced further development of the hoard.

A hoard descriptor was stored with a layout shown in Figure 5.3, where:

- *lsa* indicates the local store address of the object described, or zero if not loaded;

- *ea* indicates the effective address of the object in main memory, or zero if not allocated; and
- *size* indicates the number of bytes that the object uses.

Figure 5.3: hoard descriptor layout with ea and size fields

The blank portion of the descriptor is otherwise unused space and is able to be used for other flags — including locks, dirty/use tracking, and for other purposes detailed later in the thesis.

The use of zero in the ea field to indicate unallocated data and the presence of the size field were both adopted directly from the Burroughs descriptor, but are features that were later removed. The use of zero in the ea field allowed on-demand allocation of memory for pages, but it was found to incur a relatively large time penalty as a result of the synchronous system calls required — a large cost with no tangible benefit for the programs being tested.

It was also decided that data pages should have a fixed size, as the complexity and time overhead of transferring small allocations appeared to be strongly offset by the simplicity of managing fixed-size pages. Pages with power of two sizes are simple to locate and manipulate using shift instructions.

As all pages were the same size and there was no need to perform dynamic memory allocation, the size field was not required and was removed.

5.5 Creating descriptors

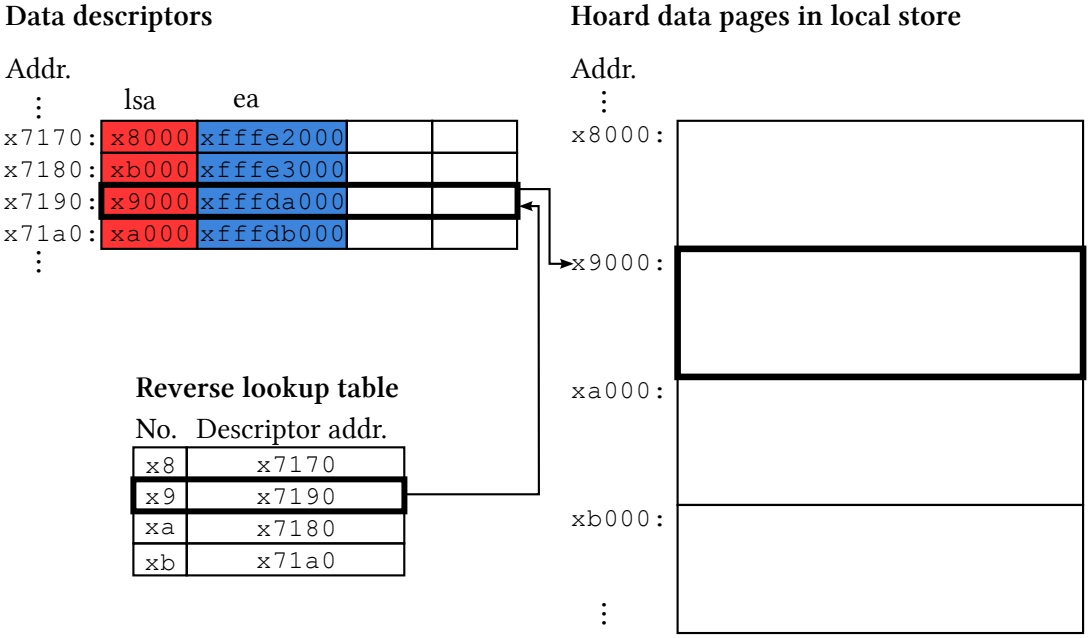
In this early iteration of the hoard, it was decided that there would be one descriptor per memory allocation.

As SPEs run only one program context at a time, have no support for protecting access locally, and limit single DMA transfers to a maximum of 16KB, segmentation was not considered to be a desirable feature.

The hoard maintains a reverse-lookup table that contains a pointer to the descriptors for each page in the hoard. This is needed to be able to locate the descriptor for a given page when that page is to be replaced, or for other actions that affect the flags for a page.

The reverse lookup table is initialised with a pointer to a special empty page placeholder descriptor, which has the advantage of not requiring a special case condition in page replacement routines to handle pages being empty.

Figure 5.4 illustrates the connections between the various details of the hoard — a data descriptor describes a page, containing the address for the page in the hoard's storage. To locate the descriptor for a given page, the reverse lookup table is used.



In this diagram, the descriptor at `x7190` contains the address of the page in hoard storage — `x9000`. The address of the descriptor for that page is determined from the page address (in this case pages are `x1000` bytes long), so the address of the descriptor is stored in the reverse lookup table location `x9`.

Figure 5.4: Reverse lookup for descriptors

For allocations larger than a single page, a descriptor was created for each page of the allocation. These were created and stored in a region of local store, intended to remain fixed in place for the duration of the program execution.

The limitations of keeping descriptors fixed in local store quickly become apparent — the most pressing limit being that maximum addressable space is (*number of descriptors* × *page size*).

For example, consider a program that has 128KB of local store available for hoard operation, and half of that (64KB) is reserved for actual program data caching, illustrated in Figure 5.5.

Table 5.1 shows the consequent maximum address space size for $65536/16 = 4096$ de-

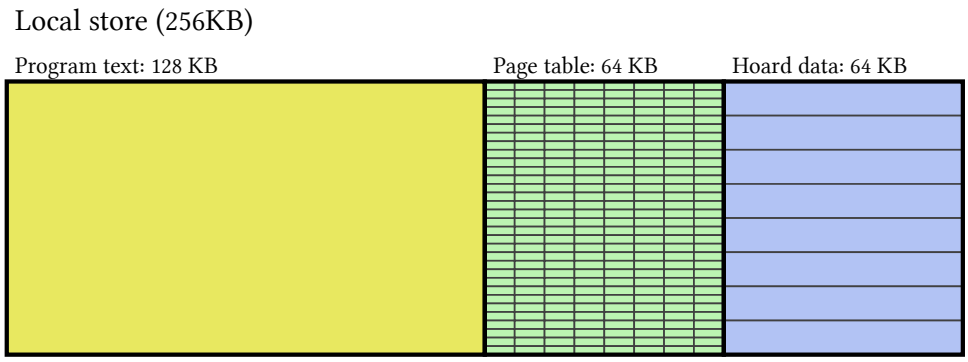


Figure 5.5: SPE local store space usage — simple page table

scriptors.

The maximum address space than can be achieved with this design is a mere 64MB. Even on the PlayStation 3 there is likely to be more RAM available for running programs, and much more virtual address space.

The amount of space available for descriptors could be increased, but this will further decrease the amount of space used to store data pages, and it cannot be assumed that even this much local store space will be available for hoard use. The design clearly needs to be modified to increase the total addressable space.

Page size (KB)	*Address space (MB)*
1	4
2	8
4	16
8	32
16	64

Table 5.1: Address space size for a range of page sizes, with a limit of 4096 descriptors

5.6 Indirection

"All problems in computer science can be solved by another level of indirection ..."

David Wheeler

"... except for the problem of too many layers of indirection."

Kevlin Henney's corollary

By adding an extra level of indirection, the size of the address space can be increased —

descriptors can be stored in memory pages of their own (descriptor pages, which will be referred to hereafter as d-pages), and managed like any other data page.

A descriptor for each d-page is kept in local store for the duration of the program run. The d-page descriptor's ea field contains the same address stored in the first descriptor on the d-page. The relationship between d-page descriptors and d-pages is illustrated in Figure 5.6.

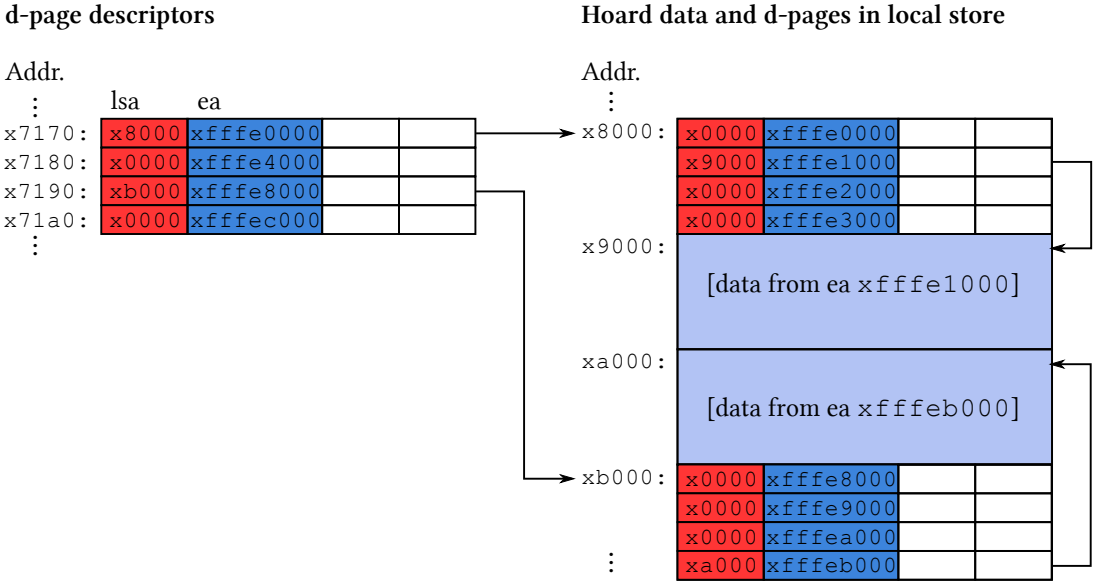


Figure 5.6: d-page descriptors and d-pages

5.6.1 How many bytes?

To compare with the estimates presented in Table 5.1, consider a similar scenario — 128KB of local store available for hoard use, but this time with only 16KB reserved for the fixed descriptors (d-page descriptors), which means that there may be $16KB/16B = 1024$ d-page descriptors.

The address space size may be defined as $1024 \times (page\,size/16) \times page\,size$ or $64 \times page\,size^2$. Figure 5.7 shows more space available for program data, and Table 5.2 shows that the maximum address space size for various page sizes is much improved. This is clearly a lot more promising. Smaller page sizes still yield small address space sizes, but there is now the possibility of working with real-world datasets and accessing an entire 32 bit address space.

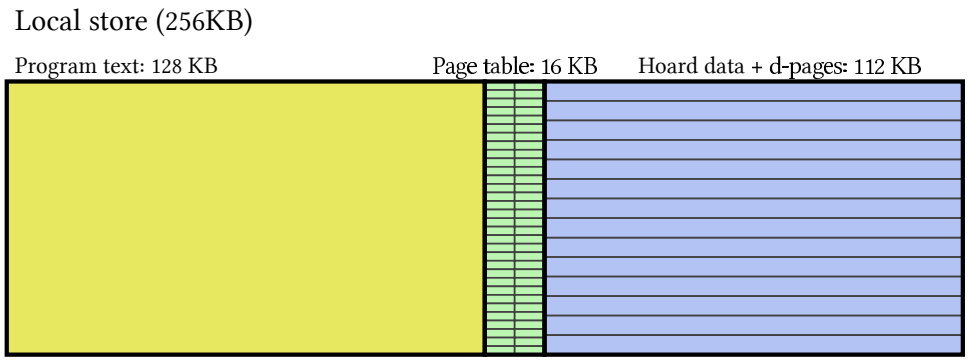


Figure 5.7: SPE local store space usage — smaller page table

Page size (KB)	*Address space (MB)*
1	64
2	256
4	1,024
8	4,096
16	16,384

Table 5.2: Address space size for a range of page sizes, with a two level lookup

There are shortcomings with this approach. Changing the memory layout gave the advantage of increasing the address space size while permitting a reduction in the amount of local store set aside for fixed descriptors. This appears to increase the space available for storing data but this may not be the case in practice. For every data page in local store, there must also be a d-page holding its descriptor. This gives a worst case situation — where every data page is described by a different d-page — that half of the pages are filled with d-pages, reducing the capacity of the hoard for storing data (see Figure 5.8).

The number of d-pages needed at any one time is dependent upon the memory access pattern of the program that is running.

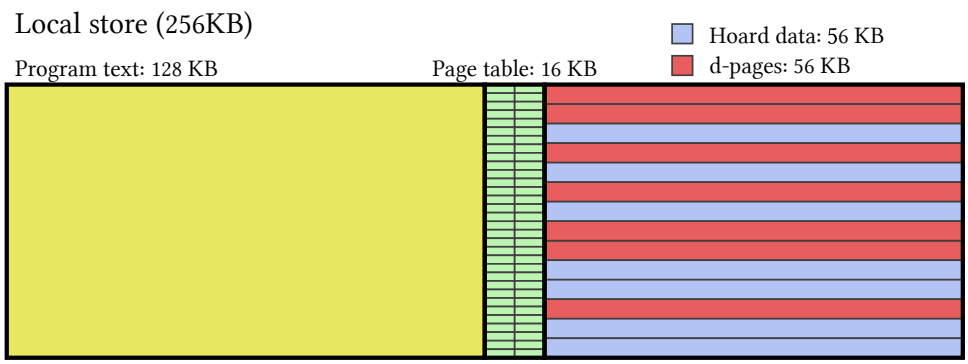


Figure 5.8: SPE local store space usage — worst case with d-pages

5.6.2 An implementation detail or two

A d-page can not be removed from local store while any descriptor on it is in use — that is, while the data page it describes is in local store. This means that d-pages need to be locked in local store while any described data pages are in local store. This locking is achieved by storing a usage-count as part of the descriptor for the d-page. An eight bit section of the *flags* section of the descriptor (as depicted in Figure 5.9) is set aside for this usage count, allowing up to 256 data pages to be loaded from a d-page. This is able to handle up to 255 loaded pages for a given d-page, which is sufficient for the highest page count configuration of 1KB pages.

Figure 5.9: data descriptor and d-page descriptor with usage counter

(The inclusion of a lock counter was included in an early revision for data pages, which was incremented before a data page was accessed and decremented when the access was complete. This ensured that a reference to data stored in a hoard-managed location is not made invalid during its lifetime. This added a performance overhead, and was removed by ensuring a suitable safe ordering through the use of a proxy object — see Chapter 6 for more information)

With a second level of lookup, accessing an address managed by the hoard works in the following way:

1. Based on the address, load d-page descriptor from local store.
2. If the lsa field of the d-page descriptor is not set, find a free page and load the d-page from main memory.
3. Increment the use-count of the d-page.
4. Based on the address, load the data page descriptor from the d-page.
5. If the lsa field of the data page descriptor is not set, find a free page and load the data page from main memory.
6. Increment the use-count of the page.
7. Based on the address, load the data from the data page.

8. Return the data to the user.

It was mentioned previously that the hoard used a FIFO mechanism for selecting pages to be replaced, but d-pages with a lock-count greater than zero cannot be replaced, as they contain descriptors of currently loaded data pages. As such, d-pages that have non-zero lock-counts are moved to the back of the queue. That a d-page is locked is a clear indication that it is in use, and so it not a good candidate for replacement in the near future. This could, perhaps, be considered a very crude extension the FIFO replacement algorithm. It makes little difference in practice.

A d-page may only be replaced in the hoard if the lock-count of its descriptor is zero, which means that d-pages may be replaced only when all descriptors on the d-page are in an un-loaded state, such that all lsa fields are zero, and page use-counts are zero.

In an early iteration of the design, d-pages were generated and stored in main memory, transferred to local store as needed. Taking into account that there are no changes to a d-page that persist beyond its time in local store, two optimisations become apparent:

1. As d-pages are not modified (there is no on-demand memory allocation), there is no need to write them back to main memory.

2. Because a d-page descriptor contains the starting address, and descriptors maintain no interesting information when outside local store, d-pages are able to be generated on-demand by the SPE, and so their transfer and storage in main memory may be eliminated.

Generating d-pages consists of an addition and a store instruction for each descriptor on a d-page, and is able to be executed faster than the time necessary to transfer d-pages via DMA. Figure 5.10 shows this process.

```
page_descriptor = descriptor(0, ea, size, 0)
offset = descriptor(0, page_size, 0, 0)
for(i=0; i < descriptors_per_dpage; ++i)
{
    dpage[i] = page_descriptor
    page_descriptor += offset
}
```

Figure 5.10: d-page generation

5.7 The need for better memory management

This hoard, as it was up to this point, worked well for programs that made a small number of allocations, and so needed only a small number of d-page descriptors fixed in local store. For a program that performed many thousands of small allocations, such as `179.art` (from the SPEC2000 suite — see Chapter 7 for more details on programs tested), local store was quickly exhausted by the large number of d-page descriptors required to describe each little memory region.

To remedy the situation, larger, multiple-page-sized memory pools are allocated from system memory, a d-page descriptor is generated for the pool, and smaller allocations are then handled from the pool.

Continuing with this idea, and with the understanding that the hoard is no longer doing any kind of management of individual objects, creating a d-page descriptor table that covers a particular portion of the system's virtual address space was considered and experimented with, but ultimately discarded due to the inflexibility of not being able to control the virtual address range provided by the system, and the inability to create a large enough table in local store particularly for small page sizes.

A solution was adopted whereby large objects are treated as independent allocations, and smaller objects are allocated from larger allocation pools.

In implementing this change to allocation, it became clear that to provide the necessary flexibility in placement, size efficiency and lookup speed, a virtual addressing method would be needed for the hoard. Hoard addresses consist of a d-page descriptor index, the index of the page descriptor in a d-page and a page offset — the offset of the requested data in the data page itself — packed together into 32 bits. One particular advantage of packing these components together is that pointer arithmetic may function without further intervention. Overflow from one part affects the next greater part appropriately.

For different page and address size configurations, it is also possible to store an index in the first part, rather than the local store address. This would have the advantage of allowing for a larger address space (potentially all the way to 32 bits) while requiring an extra register to store the base address of the d-page descriptor table.

The size of the page offset depends on the size of the data page, being from 10 bits for a 1KB page, to 14 bits for a 16KB page. The d-page index also depends on the size of data pages but as descriptors are 16 bytes each, the last four bits will always be zero and so do

not need to be stored. So the d-page index will require from 6 to 10 bits. The remaining bits serve as an index into an array of d-page descriptors. The hoard address format and lookup process is illustrated in Figure 5.11.

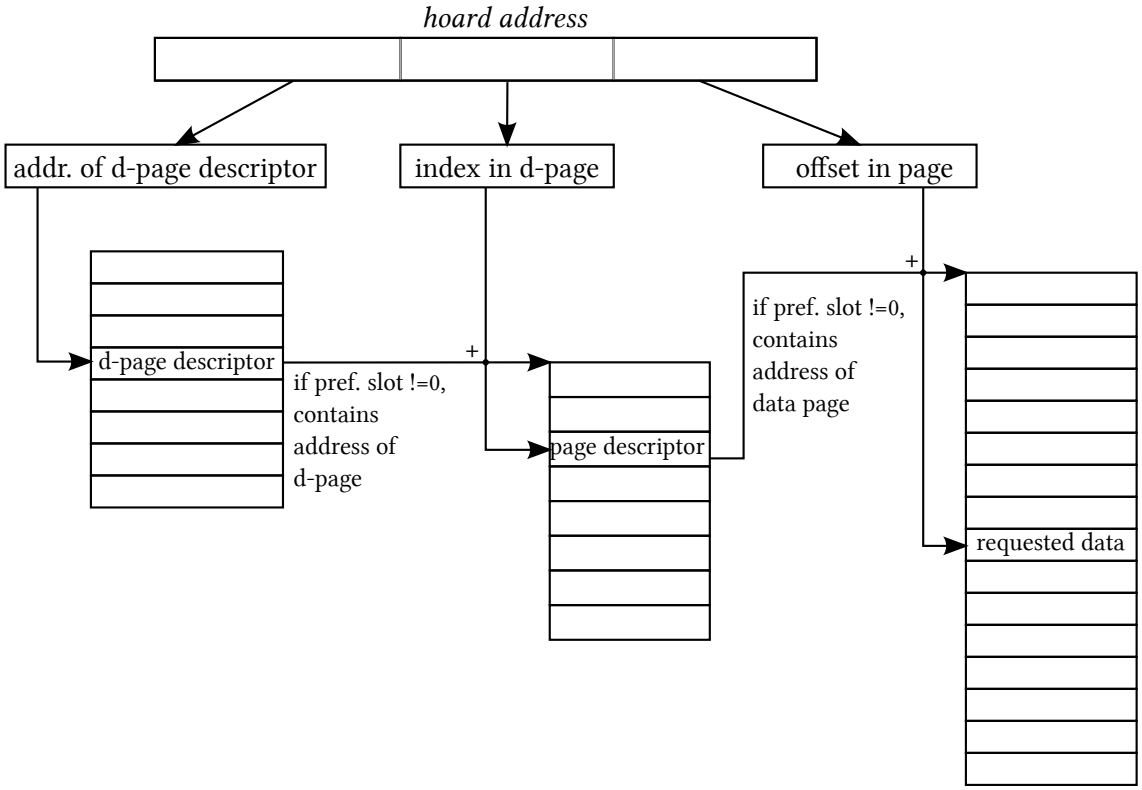


Figure 5.11: Hoard address format and mapping to a local store address

5.8 Segregation

Another change made in the development of the hoard was to separate d-page storage from data page storage. Data pages and d-pages are not the same and it is neither necessary nor ideal to have them both the same size. By keeping d-pages the same size as data pages, the total addressable space may be limited (in the case of 1KB pages) or needlessly expanded for 16KB pages, reducing space available for storing data.

Figure 5.12 shows an example of this separation and reduction in size of d-pages.

Ideally, there should be only the number of d-pages necessary to describe the data pages present in local store, but when using a FIFO replacement policy it is possible that extra unused (stale) d-pages remain present in local store, until they are replaced by the regular FIFO process.

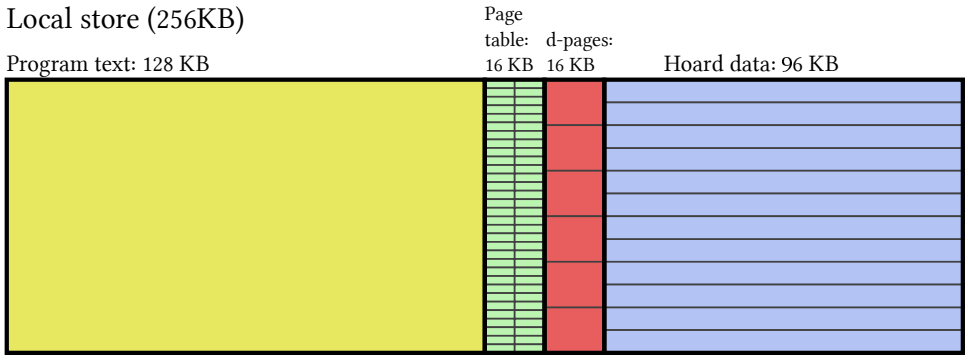


Figure 5.12: SPE local store space usage — example of separate d-page storage

Storing the two page types separately makes it easier to manage replacement in a fast, efficient fashion.

The hoard memory layout was changed such that d-pages are stored at the start of the hoard's memory region, in an area that is expanded as necessary. When a new d-page is needed, all d-page slots are checked to see if there is another d-page that may be replaced. If all slots contain in-use d-pages, space for another d-page is set aside. This may cause a data page to be ejected.

When d-pages are stored separately, there is no need to check that data pages are locked when looking for a page to replace — data pages are never locked, which means that the FIFO replacement policy will be able to always validly choose the next page for replacement, quickly and simply.

Figure 5.13 shows the addition of a *containing page* ("cont.") field, which was added to both page and d-page descriptors. This field is used to map a descriptor back to its containing d-page for the purpose of incrementing and decrementing page use counts. This is necessary because the memory allocations of d-pages are more tightly packed together, and are not power-of-two aligned like the data pages.

Figure 5.13: d-page descriptor with cont. field

The separating of d-pages from data pages also prompted a rethink of the size of the address space supported by the hoard. The decision was made to allow direct addressing of up to 256MB of memory, and the reasoning was that for the PlayStation 3, this is all the physical memory that is directly available. While Linux on the Cell BE supports virtual

memory with swapping to disk[2], allowing for a much greater storage, the goal of performance has been chosen as being more important than generality in this case. The page and d-page sizes used in this thesis when considering the hoard are shown in Table 5.3. It is worth noting that the smallest page size (1,024 bytes) requires d-pages that are four times larger to achieve the desired address space. The worst case, where each page has a separate d-page, will result in the capacity of the hoard for data being dropped to one fifth.

The size of the address space is determined by the selection of maximum d-page descriptor count, d-page, and data page sizes at compile time.

Page size (B)	*d-page size (B)*
1,024	4,096
2,048	2,048
4,096	1,024
8,192	512
16,384	256

Assumes a first level page table containing 1,024 entries.

Table 5.3: Page and d-page sizes for 256MB addressable space

5.9 Why not model a hardware cache?

The IBM SDK for Multicore Acceleration includes a software cache implementation that works a lot like a hardware cache. It may be configured as a direct mapped or four-way set associative cache, with FIFO replacement within sets. Eichenberger et al. (2006) includes a clear illustration of the function of this kind of software cache.

Figure 5.14 shows source code for the SDK cache lookup when configured with 1KB cache lines and a total size of 128KB. Note that the source code presented here has been edited to improve readability — it is functionally identical to that supplied in with the SDK.

Each set index in this cache consists of a sixteen byte vector containing four 32 bit memory addresses. When a location is requested, the requested address will be copied into all four slots of a vector (using the shuffle bytes — `shufb` — instruction) and then compared with the set index in parallel. These are quite efficient operations, but a large time penalty arises when seeking to determine which (if any) of the members of a set matched the desired address. There is no SPE instruction to identify the match to a comparison and move it to the preferred slot, so several are required.

[2]or even to video RAM.

```
1  /* Lookup EA in set directory. */
2  vec_uint4 cache_set_lookup (int set, vec_uint4 ea4) {
3      vec_uint4 dir = spu_and(*(vec_uint4 *)(cache_dir + (set << 2)),
4                              ~0x3ff|1);
5      vec_uint4 ea4_valid = spu_or (ea4, 1);
6      return spu_gather (spu_cmpeq (dir, ea4_valid));
7  }
8
9  /* Lookup a line given a set */
10 int cache_line_lookup (int set, vec_uint4 ea4) {
11     vec_uint4 exists = cache_set_lookup (set, ea4);
12     if (likely (spu_extract (exists, 0) != 0))
13     {
14         int idx = 3 - spu_extract(spu_sub((uint)31,
15                     spu_cntlz(exists)), 0);
16         return (set << 2) + idx;
17     }
18     return -1;
19 }
20
21 /* Look up EA and return cached value. */
22 qword lookup_qword (unsigned int ea)
23 {
24     unsigned int ea_aligned = ea & ~0x3ff;
25
26     /* Hash EA and compute its set number. */
27     int set  = ((ea >> 10) ^ (ea >> 11)) & 0x1f;
28     int lnum = cache_line_lookup(set, spu_splats(ea_aligned));
29     int byte = ea & 0x3ff;
30     qword ret = *(qword*)&cache_mem[(lnum<<10)+byte];
31     if (unlikely (lnum < 0))
32     {
33         lnum = cache_rd_miss_sync(ea_aligned, set);
34         return *(qword*)&cache_mem[(lnum<<10)+byte];
35     }
36     return ret;
37 }
```

Figure 5.14: SDK cache lookup of qword — implementation

Figure 5.15 shows the instructions that are required for every access to data stored in the SDK cache. At the top of the table are constant generation and address loading instructions which are necessary, but may be performed once and reused for subsequent accesses. The remaining items in the table show the instructions needed, along with the **minimum number of cycles that are required** to reach that point[3]. The list of instructions does not include possible saving and restoring of registers to the stack, branch hints or no-op instructions as these depend other code that may be scheduled in and around the cache access.

The final load instruction in this sequence is issued after **44 cycles** of processing, for **24**

[3]The number of cycles is calculated by analysing the generated assembler for the function with the spu_timing program distributed as part of the SDK.

instructions in addition to the load itself.

Figure 5.16 shows a program listing for performing lookup of data in hoard-managed storage. Figure 5.17 shows the instructions generated by the compiler for this code, indicating **a minimum of 8 instructions taking 17 cycles** before being able to issue the fetch instruction for the requested data.

It is clear that the hoard has a much lower overhead to locate data. In addition, the hoard has a fully associative storage that avoids the conflict misses inherent to a set associative cache design.

Cycle	*Insn.*	*Operands*	*Comments*	*Line no.*
–	`il`	`$2,-1024`		24
–	`ila`	`$13,cache_dir`	Base address of cache directory	3
–	`ila`	`$14,66051`	Shuffle pattern for splats	28
–	`il`	`$10,-1023`		4
–	`il`	`$80,1023`		29
–	`ila`	`$81,cache_mem`	Base address of cache storage	30
0	`rotmi`	`$16,$3,-11`	Calculate set no.	27
1	`rotmi`	`$15,$3,-10`	Calculate set no.	27
2	`and`	`$2,$3,$2`	Calculate ea of page	24
5	`xor`	`$4,$16,$15`	Calculate set no.	27
7	`andi`	`$4,$4,31`	Calculate set no.	27
7[a]	`shufb`	`$8,$2,$2,$14`	Make four copies of addr. in reg.	28
9	`shli`	`$12,$4,4`	Convert set no. to addr. offset	3
11	`ori`	`$9,$8,1`	Add VALID bit to desired page addr.	5
12	`shli`	`$8,$4,2`	Set number scaled to dir. entry	16
13	`lqx`	`$11,$12,$13`	Load cache dir. entry	3
19	`and`	`$7,$11,$10`	Mask out control bits apart from VALID bit	3
21	`ceq`	`$5,$7,$9`	Compare dir. entries with copies of sought ea	6
23	`gb`	`$5,$5`	Collect lower bits of comparison entries	6
27	`ori`	`$6,$5,0`	Copy register contents (possibly redundant)	12
29	`brz`	`$6,.L2`	Branch to miss handler if not present	12
30	`clz`	`$22,$5`	Count leading zeroes of gathered bits	15
32	`sfi`	`$21,$22,31`	(31 − leading zero count); find matching bit	14
33	`and`	`$80,$3,$80`	Calculate byte offset in page	29
34	`sfi`	`$20,$21,3`	(3 − matching bit no.); find index within set	14
36	`a`	`$19,$20,$8`	Calculate actual dir. entry	16
38	`shli`	`$18,$19,10`	Calculate offset of page in cache storage	30
39	`cgti`	`$17,$19,-1`	Check if the line was found in the cache	31
42	`a`	`$3,$18,$80`	Calc. offset of requested data in cache storage	30
44	`lqx`	`$3,$81,$3`	Load requested data	30
45[b]	`brnz`	`$17,L4`	Branch to miss handler if not present	31

[a] The `andi` and `shufb` instructions share the same cycle as they are suitable for dual-issue in the SPEs two instruction pipelines.

[b] This conditional branch to the miss handler is after a speculative load of the data has been issued. This branch is not taken in the case of a hit. In case of a miss, the data is re-loaded miss handler.

Figure 5.15: SDK cache lookup of qword — generated assembly

```
1  qword lookup_qword(uint uaddr) {
2      // extract parts from uaddr
3      dpd_t* dpage_p = (dpd_t*)(uaddr >> 14);
4      dpd_t dpage = *dpage_p;
5      const size_t b = (uaddr >> 6) & 0xff0;
6      const size_t c = uaddr & 0x3ff;
7
8      d_t page = *(d_t*)(dpage.lsa() + b);
9
10     if(likely(dpage.lsa()))
11     {
12         if(!page.lsa())
13         {
14             // page was not loaded - fetch it.
15             d_t* page_p = (d_t*)(dpage.lsa() + b);
16             fetch_page(page_p);
17             page = *(d_t*)page_p;
18         }
19     }
20     else
21     {
22         // dpage for desired page is not loaded
23         // generate it and fetch data page
24         generate_dpage((dpd_t*)dpage_p);
25         dpage = *dpage_p;
26         d_t* page_p = (d_t*)(dpage.lsa() + b);
27         fetch_page(page_p);
28         page = *(d_t*)page_p;
29     }
30     return *(qword*)(page.lsa() + c);
31 }
```

Figure 5.16: Hoard lookup of qword — implementation

Cycle	*Insn.*	*Operands*	*Comments*	*Line no.*
-	il	$2,4080	Load constants into registers	5
-	il	$83,1023		6
0	rotmi	$82,$3,-14	Rotate to extract d-page descriptor address	3
1	rotmi	$81,$3,-6	Rotate to extract d-page offset of page descriptor	5
2	and	$83,$3,$83	Mask to extract page offset of data	6
4	lqd	$80,0($82)	Load d-page descriptor	4
5	and	$81,$81,$2	Mask to extract d-page offset of page descriptor	5
10	lqx	$2,$80,$81	Load page descriptor	8
11	brz	$80,.L2	If d-page descriptor lsa not set, branch to handler	10
16	brz	$2,.L4	If page descriptor lsa not set, branch to handler	12
17	lqx	$3,$2,$83	Load requested data	30

Figure 5.17: Hoard lookup of qword — generated assembly

5.10 Fast virtual memory

Disk-backed virtual memory introduces tens or hundreds of thousands of processor cycles of latency when a required page is not in memory(Tanenbaum 2007). The SPE has a comparably low latency to retrieve data from main memory of only a few hundred cycles (Gschwind et al. 2006). For this reason, methods that are suitable and beneficial for disk-backed virtual memory systems are not suitable for use on the SPE.

Page compression (mentioned in Section 2.1.5) is unlikely to be useful. The LZO compressor (Oberhumer 2008), used by compcache (Gupta 2010) is considered to be a fast compression and very fast decompression algorithm. The peak decompression rate quoted on Oberhumer (2008) is quoted as being only one third of the speed of `memcpy()`, which is a more accurate point of comparison to the hoard as it transfers data between memory and the processor.

It seems improbable that a suitable, general purpose compressor will be able to improve performance and increase effective page capacity for the hoard. Compressors/decompressors do have a place on the SPEs in specialised cases, particularly where data may be procedurally generated. The method used by the hoard to generate d-pages (Figure 5.10) can be considered a simple example of this.

Reducing the number of accesses to disk is critical to maintaining high performance when using disk for page storage. Many page replacement methods perform work on each access to a page and when selecting a page for replacement to try to reduce the number of replacements. The large cost of a miss makes this extra work beneficial. For the SPE, misses take much less time to handle and so the extra work may not be able to pay for itself.

5.11 Reinventing the wheel

[Talking about the wheel, the "single simplest machine in the entire universe":]
'All right, Mr Wiseguy,' she said,
'you're so clever, you tell us what colour it should be.' (Adams 2002, p. 373)

"... any algorithm that used to be designed for a disk 20 years back is probably a good algorithm to be used on memory today" (Engel 2008)

The development of the hoard began as an attempt to implement a software managed cache for the Cell BE SPEs. The result is something that is *almost, but not quite, entirely unlike* a cache,[4]. Instead, what has been implemented (without it being the up-front intent) is a paged virtual memory system for the SPEs.

The hoard provides a fully associative storage, using FIFO page replacement. When a page is replaced, it is always written back to main memory. Page sizes from 1KB to 16KB are supported and configurable at compile time. Pages are allocated at runtime from the remaining local store after program text and stack boundaries are taken into consideration.

The next chapter describes the programming interface for the hoard in detail.

[4] "Almost, but not quite, entirely unlike tea." (Adams 2002, p. 198)

Chapter 6

The Hoard API

This chapter describes the hoard application programming interface — the part of the hoard that interfaces with a program directly.

The program-facing interface of the hoard is designed to make it easy to add managed memory accesses to a program that has been written assuming a flat, large address space, and to reduce the chance of introducing errors to a program in the process.

The first part of this chapter introduces the hoard implementation, particularly the guiding principles, and the use a `hoard_ptr` object in place of a simple pointer. With this, the details of some of the more complex problems associated with the implementation are examined.

After this, there is some detail of the process involved in modifying an existing program to use hoard managed storage; this demonstrates how the hoard design facilitates compiler-assisted conversion. The configuration of the hoard is presented, and the last part of the chapter provides a comparison between the techniques used to implement the hoard and alternatives, including those of other software-cache-like systems.

6.1 The implementation of the hoard user interface[1]

Two overarching objectives guide the implementation of the hoard interface:

- *Correctness* — it should be difficult to misuse hoard-managed storage; and

[1]Code fragments contained in this chapter are to be considered indicative rather than exact reproductions of hoard source code. Technicalities that are not relevant to this explanation will be omitted. More complete C++ header contents may be found in Appendix A.

- *Compiler-assisted usage* — if there is a way that some part of the system can be misused, the incorrect usage should generate a compile-time error or warning.

Balanced with these objectives is the desire to provide an abstraction that is able to directly replace a regular pointer, handling every operation that a regular, simple C pointer can handle, including arithmetic and indexed array access.

The hoard interface is implemented within a C++ header that defines a templated data structure, `hoard_ptr`, that is the fundamental pointer data structure used anywhere that an SPE program requires hoard-managed storage.

The `hoard_ptr` structure has a number of overloaded operators, allowing it to be used in place of a regular pointer without requiring many changes to existing programs. `hoard_ptr` attempts to support all regular pointer operations in a way that allows it to be a drop-in replacement for regular pointers. Differences between `hoard_ptr` behaviour and that of regular C pointers will be noted where they are known.

In the sections following, pointer operations and their implementation for the `hoard_ptr` type will be considered.

6.1.1 Declaration

In C, declaring a pointer is done in the form `T*`, for some type `T`. When a pointer to hoard-managed data is required, `hoard_ptr<T>` may be used. Typedefs are defined for all primitive types in the form `T_p`. For example, a regular pointer to int would be declared as `int* ip`, the `hoard_ptr` equivalent would be `hoard_ptr<int> ip`, or `int_p ip`.

For hoard-managed arrays of hoard-managed data, `T_pp` is a typedef of `hoard_ptr< hoard_ptr<T> >`, and `T_ppp` is similarly defined. Figure 6.1 has a side-by-side comparison of the use of regular and hoard managed types.

It is possible to define combinations of local and hoard-managed pointers. For example, a local pointer to a hoard managed region would be declared as `hoard_ptr<T>*`, and a hoard-managed array of local pointers would be `hoard_ptr<T*>`.

6.1.2 Construction and assignment

A pointer is not much use unless it points to something. `hoard_ptr<T>` supports the following forms of construction and assignment:

Regular C:

```
uchar **cimage;
double **tds;
double **bus;

typedef struct {
    double *I;
    double W;
    double X;
} f1_neuron;


f1_neuron *f1_layer;
```

Hoard:

```
uchar_pp cimage;
double_pp tds;
double_pp bus;

typedef struct {
    double_p I;
    double W;
    double X;
} f1_neuron;

typedef hoard_ptr<f1_neuron> f1_neuron_p;
f1_neuron_p f1_layer;
```

Figure 6.1: Use of hoard pointer types

6.1.2.1 Default construction

```
hoard_ptr<T>() {}
```

No value assigned, pointer target is undefined.

6.1.2.2 Copy construction or assignment from `hoard_ptr<T>`

```
hoard_ptr<T>(const hoard_ptr<T>& r) : ha(r.ha) {}
```

Target address copied from source `hoard_ptr`.

6.1.2.3 Explicit construction from `ha_t`

```
explicit hoard_ptr<T>(ha_t ha_) : ha(ha_) {}
```

Used internally by the hoard, this constructor allows initialisation of a `hoard_ptr` from the hoard address type (described in Section 5.7). `ha_t` is a typedef of an integral type and so this constructor is explicit to prevent accidental, silent conversion of some integral value to a `hoard_ptr`.

6.1.2.4 Construction or assignment from zero

```
struct null_ptr { null_ptr(null_ptr*) {} };
hoard_ptr<T>(null_ptr) : ha(0) {}
hoard_ptr<T>& operator=(null_ptr) { ha = 0; return *this; }
```

Using a specially defined `null_type` to avoid construction or assignment from arbitrary integer values, a `hoard_ptr<T>` may be set to zero.

6.1.2.5 `hoard_ptr<void>` specialisation

`hoard_ptr<void>` is specialised with reduced functionality compared to `hoard_ptr<T>`, analogous to the limitations on the `void*` type. It supports construction and assignment in the same way as the more general `hoard_ptr<T>` implementation, but also supports construction from `hoard_ptr<T>`, with arbitrary type `T`, as may be done with regular pointers. i.e.

```
int* ip;
void* vp = ip; // valid
char* cp = ip; // error
hoard_ptr<int> ih;
hoard_ptr<void> vh = ih; // valid
hoard_ptr<char> ch = ih; // error
```

6.1.3 Dereferencing

```
ref_t operator*();
ref_t operator->();
ref_t operator[](int32_t);
```

As much fun as twiddling pointers can be, they aren't useful unless the programmer can get to what is pointed at. In the C and C++ programming languages, the unary `*` operator is used to dereference a pointer and access the target, and C++ allows overloading of that operator for a class.

`hoard_ptr<T>` has explicit implementations of operator `*`, the member-by-pointer operator `->`, and the array subscript operator `[]`, each of which performs a request to the hoard storage for the data on the page requested, loading a page from main memory if needed and returning a reference to the data[2].

These three operations are the only `hoard_ptr` operators that attempt to access hoard storage. Dereferencing an invalid `hoard_ptr<T>` is undefined behaviour.

[2] or a proxy reference type, described in Section 6.2

It is not possible to dereference a `void*`, and for this reason the `hoard_ptr<void>` specialisation does not implement these operators.

6.1.4 Comparison

```
/* based on boost's operator bool */
typedef ha_t hoard_ptr::*unspecified_bool_type;
operator unspecified_bool_type() { return ha ? &hoard_ptr::ha : 0; }

bool operator==(const hoard_ptr<T>& r) { return ha == r.ha; }
bool operator!=(const hoard_ptr<T>& r) { return ha != r.ha; }
bool operator< (const hoard_ptr &r)    { return ha <  r.ha; }
bool operator<=(const hoard_ptr &r)    { return ha <= r.ha; }
bool operator> (const hoard_ptr &r)    { return ha >  r.ha; }
bool operator>=(const hoard_ptr &r)    { return ha >= r.ha; }
```

There are a number of meaningful comparisons that may be performed with pointers — this include comparison against zero to determine whether a pointer is set or not, and comparing the relative ordering or equality of two pointers. As is the case in C and C++ comparing pointers is unspecified for two pointers that do not point to the same object (e.g. ISO/IEC (1998)).

For testing the validity of a pointer, the `hoard_ptr` class uses the "safe bool idiom" (Karlsson 2004) to prevent accidental conversion from a `hoard_ptr` to an integral value.

6.1.5 Arithmetic operators

```
hoard_ptr  operator+ (int32_t s);
hoard_ptr  operator- (int32_t s);
hoard_ptr& operator+=(int32_t s);
hoard_ptr& operator-=(int32_t s);
hoard_ptr& operator++();
hoard_ptr  operator++(int);
hoard_ptr& operator--();
hoard_ptr  operator--(int);
ptrdiff_t  operator- (const hoard_ptr& r);
```

`hoard_ptr<T>` supports pointer arithmetic through the `++`, `--`, `+` and `-` operators as is the case for regular pointers.

6.2 Access ordering via proxy object

When a `hoard_ptr` appears on both sides of an expression, problems may arise. Each dereferencing access to memory through a `hoard_ptr` may have side effects that invalidate any existing references to hoard-managed memory locations.

Consider the expression `*a = *b`, where `a` and `b` are both `hoard_ptr`s. In evaluating this expression, the following steps take place:

- Determine the local store address of the memory location pointed to by `a`. Call this `lsa(*a)`, being the local store address of `*a`. If `lsa(*a)` is not defined — that is, there is no copy of `*a` in local store, load the page containing `*a` into the hoard.
- Determine the local store address of the memory location pointed to by `b`, i.e. `lsa(*b)`. If `lsa(*b)` is not defined, load the page containing `*b` into the hoard.
- Load the value of `*b` from `lsa(*b)` into a register and store it to `lsa(*a)`, i.e. `*lsa(*a) = *lsa(*b)`

There are constraints here that need to be carefully addressed. The loading of `*b` into local store may cause `lsa(*a)` to become invalid, if the memory page that contains `lsa(*a)` is replaced for `*b` to be loaded.

This is a problem that may be solved through appropriate ordering of the above steps — C/C++ does not require that the left hand side of an expression be calculated before the right hand side (unlike Java, for example). But it is the case that GCC generates code to determine the target memory location for the left hand side of the expression before the value to be assigned on the right hand side of the expression is evaluated. During the development of the hoard this problem was experienced, causing incorrect program results.

As a result, it is necessary to defer the determination of `lsa(*a)` until after the right hand side of the expression has been fully evaluated, and only the assignment remains.

To achieve this, the operators `*`, `->` and `[]` of `hoard_ptr` do not return a `T&`, or even check for the presence of requested data in local store. Instead, these operators return a

proxy object of type `hoard_ref`, described in the next sub-section.

6.2.1 `hoard_ref`

This class is a proxy designed to ensure correct ordering of assignments to hoard-managed memory locations. This is achieved by overloading all assignment operators (=, +=, -=, *=, /=, etc) such that the right hand side value is fully evaluated and (notionally) copied out of the hoard before the memory reference of the left hand side is determined.

Consider:

```
template<typename T>
hoard_ref<T>& operator=(hoard_ref<T>& l, const hoard_ref<T>& r) {
    T rval = r; // complete the evaluation of r
    T& lval = hoard_lookup(l);
    lval = rval;
    return l;
}
```

It can be seen in this case that ordering is correct. The evaluation of the right hand side is completed and reduced to a temporary value before the memory location of the left hand side is found in the hoard. In this way, the ordering of assignment evaluation will behave correctly for hoard pointers.

The various assignment, increment, and decrement operators as well as an implicit typecast to a `const` value reference type are implemented for `hoard_ref`. The typecast allows `hoard_ref` results to be used silently in expressions not containing other `hoard_ref` operands.

6.2.2 Limitations of `hoard_ref`

`hoard_ref` is largely transparent in use, but does require a few small changes to a program using `hoard_ptr`. The need for these changes will be reported as errors or warnings by the compiler.

6.2.2.1 Compound types

When a compound type is stored in the hoard, correctly handling access to its members needs extra care. Specifically, if a program attempts to access members of a hoard-managed type via the '.' operator (member access), a compile time error will result as the `hoard_ref` will not have a member with the same type and name expected. C++ does not support the overloading of the member access operator, and so some other method is required to intercept accesses to struct members.

To work around this limitation, it is necessary for the hoard user to define a 'shadow' `hoard_ref` specialisation, mirroring the structure of the shadowed type.

For example, scanner.c from `179.art` (in the SPEC2000 suite of programs) has the following type definition:

```
typedef struct {
    double y;
    int   reset;
} xyz;
```

The following added definition provides correct functionality:

```
template<> struct hoard_ref<xyz> {
    hoard_ref<double> y;
    hoard_ref<int> reset;
    hoard_ref(ha_t ha) :
        y(ha + offsetof(xyz, y)),
        reset(ha + offsetof(xyz, reset))
    {}
};
```

The specialisation shown here provides a definition that allows `hoard_ref` to function correctly, populating a `hoard_ref<xyz>` object with proxy objects, each storing appropriately offset addresses so that memory accesses affect the correct locations. The calculation of addresses, offsets, and function calls are largely eliminated at compile time, incurring little or no additional runtime overhead[3].

[3]Unfortunately, it's also very ugly and not well suited to replacement by a single magic preprocessor macro.

This kind of mechanical definition is required for each data type for which member access is required. `hoard_ptr` supports dereferenced member access via the `->` operator without the need for the above extra definition.

Disallowing the use of the member access operator for objects managed by the hoard was considered, permitting only use of the (overloadable) -> operator. This was deemed to require more intrusive changes to existing programs than requiring specialisation of `hoard_ref`. Specialisation must be added for managed types, of which there were only a small number in the tested programs.

6.2.2.2 Untyped expressions

Some widely used C standard library functions use a variable-length argument list that doesn't provide any meaningful type safety. For example, the `printf()` and `scanf()` functions take a format string, followed by an arbitrary number of arguments. These arguments are interpreted based on the conversion specifications present in the format string, which may be arbitrarily constructed at runtime.

Accessing hoard-managed data through a `hoard_ptr` will return a `hoard_ref` object which will be converted to an appropriate type when type information is available. When attempting to pass a `hoard_ref` object to `printf()`, the compiler will issue the following warning[4]:

```
warning: cannot pass objects of non-POD type
'struct hoard_ref<foo>' through '...'; call
will abort at runtime
```

These can be identified by the programmer, and fixed by adding an explicit cast to the referenced type. For example:

```
int_p x;
printf("%d", *x); // will generate warning or error
printf("%d", (int)*x); // correct, safe
```

[4]Warning in gcc-4.4, error in gcc-4.5

6.2.2.3 Address of indexed array element

Two ways for finding the address of an array element were encountered while developing the hoard: `(foo+x)` and `&(foo[x])`. These two forms should achieve the same result, but the two are subtly different when `foo` is a `hoard_ptr<T>` object.

The first form, `(foo+x)` has a straightforward implementation, where the lookup index contained within the object is incremented by `x*sizeof(T)`, but the same is not true when trying to take the address of an indexed array access.

While it may be possible to support the second form through an appropriate overloaded `operator&()` for `hoard_ref`, this is not straightforward to implement in practice due to the dependencies between the `hoard_ptr` and `hoard_ref` classes and related limitations on overloading the unary `&` operator[5]. As such, for the sake of simplicity there is no attempt to implement this operator and all pointer arithmetic is converted to the first form. Errors will be generated if the second form is attempted.

6.3 Basic hoard configuration

There are a few options available to control hoard configuration, but the primary setting used in this research is the `HOARD_PAGE_SIZE_BITS` macro. It is used to set the size of the hoard memory page size, where page size will be $2^{HOARD\_PAGE\_SIZE\_BITS}$ bytes, and may be a value between 10 and 14, corresponding to the supported page sizes from 1KB to 16KB. The default value is 10, being 1024 byte pages.

There are other configuration options for the hoard. The `HOARD_ADDRESS_SPACE_BITS` macro controls the maximum size of the hoard address space. This remains set for this research at 28, corresponding to an address space of 256MB, the size of the main memory in the PlayStation 3.

6.4 Adopting the hoard

To facilitate the process of adding hoard memory management to existing programs, certain standard library functions have been re-implemented to work with `hoard_ptr`s, including memory allocators, file access functions, and other memory manipulators.

[5] "The address of an object of incomplete type can be taken, but if the complete type of that object is a class type that declares `operator&()` as a member function, then the behavior is undefined (and no diagnostic is required)." (ISO/IEC 1998, 5.3.1, p. 76)

By using the memory allocators — the producers of pointers — the parts of a program that need to be changed become apparent through compiler errors and warnings. It is a mechanical process to change pointer declarations from regular pointer types to `hoard_ptr` equivalents.

The hoard does require C++, but works very well for porting C code to the SPE. C is not just a subset of C++ though, and there are likely to be some problems when trying to compile the C program with a C++ compiler.

6.4.1 Memory allocators

The best place to start converting a program to hoard usage is the `malloc()` family of functions. These typically return a `void*` — the hoard provides a simple `malloc()` implementation that returns a `hoard_ptr<void>`.

When using the replacement `malloc()` functions, the compiler will produce errors when the values returned are no longer usable within the program due to type incompatibility. From here it is a mechanical process to replace regular pointer types in the program with hoard equivalents as needed to allow compilation of the program.

As well as `malloc()`, it is possible to use `calloc()`, and `mmap()` with the hoard.

```
void_p malloc(size_t size);
void_p calloc(size_t nmemb, size_t size);
void_p mmap_hoard(unsigned long long start, size_t length,
                  int prot, int flags, int fd, off_t offset)
```

6.4.2 File access

The `read()` and `fread()` functions are used to transfer data from a file to memory. The experience in developing the hoard was that this was often more than the size of local store, and often into dynamically allocated memory regions.

The hoard implementations of these functions are parametric overloads of their 'regular' counterparts, replacing the `void*` parameters with `hoard_ptr<void>` (`void_p`).

The hoard implementations are not particularly efficient, primarily due to the way that the `read()` system call is implemented by libspe2 (IBM Corporation et al. 2008d) — synchronously loading data from a file to the local store of the SPE.

```
ssize_t read(int fd, void_p buf, size_t count);
size_t fread(void_p ptr, size_t size, size_t nmemb, FILE* stream);
```

6.4.3 Memory manipulators

There are overloaded implementations of `memcpy()` and `memset()` that will work with `hoard_ptr`s.

```
void_p memcpy(void_p dest, const void_p src, size_t n);
void_p memcpy(void_p dest, const void* src, size_t n);
void*  memcpy(void* dest, const void_p src, size_t n);
void_p memset_cache(void_p s, int c, size_t n);
```

6.5 Porting example

Creating a program from scratch that makes use of the hoard can be done by replacing regular pointer types with `hoard_ptr` usage. Converting an existing program to use hoard pointers can be done piece by piece, where allocations that are too large for local store may be replaced with hoard-based equivalents. The compiler (GCC) will detect most incorrect uses of `hoard_ptr` objects and will warn or produce errors accordingly. It is the case that these errors or warnings will not clearly indicate the nature of the change required and so it may not be immediately clear to a programmer what changes are required.

While is is difficult to prove that the design of the hoard will ensure that warnings or errors are generated for all possible error conditions, it has been the case that all changes that have been required in development and testing of the hoard are correctly detected and warned about.

The benchmark program `179.art` is discussed in more detail in Chapter 7, but it serves as a useful example of modifying a program for use with the hoard.

The `protoize` program was used to generate ANSI C prototypes, making the source compilable as C++.

The option `-include hoard_basics.h` is passed to the compiler, which includes various macro definitions, typedefs, and replacements for standard library functions like `malloc()`.

When compiled, the following errors and warnings are generated:

```
In function void init_net():
246:62: error: invalid cast from type void_p to type f1_neuron*
255:55: error: invalid cast from type void_p to type double*
270:40: error: invalid cast from type void_p to type xyz*
In function void analog_conv():
286:47: warning: format %f expects argument of type float*, but
argument 4 has type double*
286:47: warning: format %f expects argument of type float*, but
argument 6 has type double*
286:47: warning: format %f expects argument of type float*, but
argument 4 has type double*
286:47: warning: format %f expects argument of type float*, but
argument 6 has type double*
In function void get_pat():
298:39: warning: format %f expects argument of type float*, but
argument 3 has type double*
298:39: warning: format %f expects argument of type float*, but
argument 3 has type double*
In function void loadimage(char*):
714:63: error: invalid cast from type void_p to type char*
720:71: error: invalid cast from type void_p to type unsigned char**
729:72: error: invalid cast from type void_p to type unsigned char*
In function void alloc_td_bu():
797:51: error: invalid cast from type void_p to type double**
798:51: error: invalid cast from type void_p to type double**
806:53: error: invalid cast from type void_p to type double*
807:53: error: invalid cast from type void_p to type double*
In function void load_train(char*, int, int):
907:65: error: invalid cast from type void_p to type char*
```

The warnings "format %f expects argument of type float*" are unrelated to the hoard and may be safely ignored.

Errors of the form "invalid cast from type void_p" are relevant. They show the locations in the program that the `malloc()` function is being called, returning a object of type `void_p` (which is a typedef for `hoard_ptr<void>`) and being assigned to a regular

pointer variable. To address these issues, the types of the variables that are assigned the return value from `malloc()` must be changed, typecasts changed, and appropriate typedefs added. For example, the following code:

```
xyz *Y;
Y = (xyz *)malloc(numf2s*sizeof(xyz));
```

is changed to the following:

```
typedef hoard<xyz> xyz_p;
xyz_p Y;
Y = (xyz_p)malloc(numf2s*sizeof(xyz));
```

This then raises a number of errors of the form "struct hoard_ref<xyz> has no member named y", which are generated by lines like

```
if ( Y[i].y > 0)
```

The `hoard_ref` template object returned by `hoard_ptr<xyz>::operator[]` has no member named y. To solve this, we must add an appropriate `hoard_ref` specialisation (as was shown in in 6.2.2.1):

```
template<> struct hoard_ref<xyz> {
    hoard_ref<double> y;
    hoard_ref<int> reset;
    hoard_ref(ha_t ha) :
        y(ha + offsetof(xyz, y)),
        reset(ha + offsetof(xyz, reset))
    {}
};
```

`179.art` uses the `fscanf()` and `printf()` functions, and both use format strings and a variable length list of parameters. Using the `-Wformat` option, GCC provides warnings wherever a hoard pointer or reference type is used as it does not match the format string.

For `fscanf()` it is necessary to add a temporary variable and an extra copy, so

```
fscanf(fp,"%f", &f1_layer[i].I[cp]);
```

may be replaced with

```
double t;
fscanf(fp,"%f", &t);
f1_layer[i].I[cp] = t;
```

For `printf()` there is a helper function `c_ref()` that performs the necessary lookup and temporary copy. Where a warning is generated for a line like

```
printf(" %8.5f ", f1_layer[i].I[cp]);
```

this will eliminate the warning (and the underlying error)

```
printf(" %8.5f ", c_ref(f1_layer[i].I[cp]));
```

A less obvious and indirectly riskier part of the conversion of this program is the use of multiples of `sizeof(double*)` passed to `malloc()`, which should actually be `sizeof(double_p)`. It is the case that a `hoard_ptr` and a regular pointer have the same size (32 bits), though during the development of the hoard there was consideration given to making `hoard_ptr` larger (64 or 128 bits), which would make this allocation too small.

Another difficulty in the use of the hoard relates to stack usage by a program. Hoard page storage is allocated to be as large as possible while leaving at least at least 12KB of stack space available for the running program, which was found to be sufficient for most programs tested. In some cases, more is required, and the hoard may be configured to leave more (through the use of the `STACK_RESERVATION` macro). Exhaustion of stack space will lead to unreliable program performance — random crashes or incorrect results.

One area in which the hoard has not been well tested is for structures that are not multiples of a power of two. If not handled with care, the hoard may not perform correctly for accesses that cross a page boundary.

Effective and safe use of the hoard does depend on the programmer having an understanding some of the more complex features of C++, and an understanding of the operation of the hoard — the function of `hoard_ptr` and related classes, and the performance consequences and limitations of the implementation.

6.6 Alternatives

In this section, consideration will be given to some alternative approaches to those taken in this implementation of the hoard.

6.6.1 The bigger picture — why compile- and run-time?

The hoard is implemented entirely outside of the compiler. It consists of headers that add the necessary checks and function calls for managed data as pointer-like classes, used at compile time, as well as functions linked to the program to provide the underlying hoard management.

Writing a cache as a compile- and run-time library will result in a certain level of intrusiveness into programs using the library, although C++ features like operator overloading allow this intrusiveness to be fairly minimal. In contrast, the IBM SDK software cache, which is implemented in C, requires an explicit macro wrapping every access to cache-managed data — a much greater intrusion into a program.

It has been possible to take the `hoard_ptr` and `hoard_ref` classes and use them with the IBM SDK cache with only minor modifications, providing a similar reduction in intrusiveness into a program using this cache.

Regarding intrusiveness, the named address space extension requires an amount of modification to a program similar to the hoard — managed data and pointer types must be annotated with an address space specifier, although conversion of the code from C to C++ is not required[6].

The use of named address spaces also requires access to a compiler that provides this feature.

A compiler-based implementation has the advantage of being able to perform greater analysis on the program's memory use at compile time. The hoard does use a number of techniques to maximise the compiler's ability to optimise the program, including heavy inlining, careful use of intrinsics and types to allow the compiler the best chance at managing program flow and memory usage. There are certain behaviours of the hoard that it is not possible for the compiler to optimise without modification, particularly the elimination of repeated, redundant accesses to managed locations.

[6]There is currently no existing standard for the use of named address spaces in C++

Implementation outside of the compiler does effectively eliminate any opportunity to apply analyse and classify memory accesses as is done in Gonzàlez et al. (2008). While it could be handled through annotation applied by the programmer and implemented via specialised `hoard_ptr` types, these options go beyond the scope of this thesis and the design goal of minimal intrusiveness into existing code.

When attempting incorrect use of the hoard, error messages generated by the compiler may not clearly indicate the cause of the problem. For example, if there is no `hoard_ref` specialisation for `struct xyz` defined (as explained in 6.2.2.1), the compiler will generate the following errors (and more) when compiling `179.art`:

```
scanner.cpp: In function 'double g(int)':
scanner.cpp:120: error: 'struct hoard_ref<xyz>' has no member named 'y'
scanner.cpp: In function 'void find_match()':
scanner.cpp:131: error: 'struct hoard_ref<xyz>' has no member named 'y'
scanner.cpp:131: error: 'struct hoard_ref<xyz>' has no member named 'y'
scanner.cpp: In function 'void reset_nodes()':
scanner.cpp:378: error: 'struct hoard_ref<xyz>' has no member named 'y'
scanner.cpp:379: error: 'struct hoard_ref<xyz>' has no member named 'reset'
...
```

There would be usability benefits from adding diagnostic support to the compiler to provide better information to a programmer. An external error processor like STLFilt (Zolman 2008) may be able to achieve a similar result.

Implementing the hoard as a software library, external to the compiler allows easy development, fast compiles, and reduced complexity. The use of C++ allows straightforward modification of existing programs, with very little intrusion. A fully software-based solution allows for maximum flexibility for modifying and tuning the implementation to specific programs.

6.6.2 More specific — `hoard_ref` specialisations

The use of `hoard_ref` objects to encapsulate hoard accesses leads to several problems, as detailed in Section 6.2. In the development of the hoard, there were several alternatives considered. These include:

- *Modifying the compiler* to evaluate expressions in a more hoard-friendly fashion;

- *Manually splitting assignments*, storing the right hand side of an expression in a temporary variable before assigning to destination; or
- *Locking pages* from being replaced when a reference is created.

6.6.2.1 Modifying the compiler

Changing GCC to evaluate the right hand side before the left was fairly straightforward and appeared to provide correct results for a range of programs, but it was decided that this change was not generally suitable — requiring a modified compiler to make use of the hoard is unreasonable for general use.

There is no need for struct specialisations or explicit type-casts with this method.

6.6.2.2 Manually splitting assignments

Identifying and splitting all potentially risky assignment expressions is a tedious and difficult task as the compiler will not be able to identify these and provide errors or warnings. Instead, the entire program must be inspected, which is tedious and error prone, and it is for this reason that this approach was abandoned.

There is no need for struct specialisations or explicit type-casts with this method.

6.6.2.3 Locking pages

Page locking may be achieved through a proxy-object approach not dissimilar to that used with `hoard_ref` — incrementing a page lock-count upon construction of the object and decrementing during destruction, and overloading the assignment operators of the proxy object to allow assignment through it.

It was hoped that the compiler would be able to avoid generating the lock increments and decrements in most cases, but when investigated this was found to not be the case and the extra increment and decrement for every hoard access resulted in an increase in program runtime.

Struct specialisations and explicit type-casts would be required for this method.

Chapter 7

Methodology

In the previous chapter, it was explained how the hoard has been designed to provide a simple, easy to use, software memory abstraction — a tool that will aid the porting of existing programs to the Cell BE SPE. In Chapter 5, the function of the hoard was presented, and it was shown that the architecture of the Cell BE suits a memory management system more closely resembling paged virtual memory than a traditional hardware cache.

This chapter describes the basic benchmarking tools and an experimental method for comparing the hoard with other SPE memory management methods, and allows comparison of the performance and practical usefulness of such systems.

7.1 Overview of benchmarks

There are a number of benchmarking tools and suites available. Of those considered, the SPEC CPU2000 suite seemed to provide a well regarded selection of programs with which to test the hoard.

The SPEC CPU2000 benchmark suite (Standard Performance Evaluation Corporation 2005) consists of a range of computational benchmarks written in various languages. It includes several programs written in C that have small program text (fitting into the SPE local store), and are designed to be run on systems with at least 256MB of RAM available. SPEC benchmarks are well understood, heavily analysed and have been run on a wide range of platforms (Standard Performance Evaluation Corporation 2005).

SPEC have a more recently released benchmarking suite in the SPEC CPU2006 collec-

tion, but this is targeted at systems with at least 1GB of RAM, which is more than is available on the PlayStation 3 (Standard Performance Evaluation Corporation 2008).

In addition to the SPEC CPU2000 benchmarks selected, several sample programs from the IBM Cell SDK for Multicore Acceleration (IBM Corporation et al. 2009a) were used throughout the development and testing of the hoard.

One extra benchmark was taken from the MediaBench suite (Lee et al. 1997) as a program that performs stream-based processing of video data seemed appropriate for consideration on the PlayStation 3.

Programs were excluded from consideration if their text size was too large for the SPE as it was considered that adding a second type of caching would be likely to obscure results from the hoard's data caching.

The following benchmarks were selected for use in this research.

7.1.1 `qsort` — quicksort

From the IBM Cell SDK, and originally written as an example of the use of the IBM SDK software cache, this benchmark is relatively simple, generating a pseudo-random set of floating point numbers and then sorting them. The same set of numbers is generated each time the program is run, so results of each run of the program are comparable. Also, the number of comparisons (or swaps) serves as a simple validation for the correctness of a program run.

As this program is provided as an example with the SDK cache, it seemed appropriate to make use of it for comparison with the hoard.

This benchmark is run with a dataset of 4,194,304 (that is, 2^{22}) vectors of single-precision floating point numbers, requiring 64MB of system memory. This is large enough to provide clear, measurable trends while limiting memory usage. This avoids contention with other processes, delivering consistent results between program runs.

7.1.2 `hsort` — heap sort

From the IBM Cell SDK, this is an example of the use of the IBM SDK software cache. It generates the same set of floating point numbers and sorts them using a heap sort implementation. The number of comparisons serves as the test of correctness. Like `qsort`, `hsort` is provided as an example use of the SDK cache and has been chosen for comparison

for that reason.

Heap sort exhibits much lower locality of reference than quicksort and so can be expected to reveal the consequences of a higher miss rate than `qsort`. Like `qsort`, this benchmark is run with a dataset of 4,194,304 vectors of floating point numbers.

7.1.3 `julia_set` — 3D fractal renderer

Figure 7.1: Images rendered with `julia_set`

From the IBM Cell SDK, a this is a texturing Julia set fractal (Weisstein 2010) renderer. This program uses all available SPEs to render a scene, and SPEs perform read-only access to texture data. The distribution of work to particular SPEs is varied dynamically from frame to frame.

Scene rendering involves numerous single-precision floating point operations and multiple texture lookups per pixel. Texture tiles are cached in local store.

This benchmark renders one hundred 1024×1024 pixel frames.

7.1.4 `179.art` — neural net based image recognition

A double-precision floating point benchmark from SPEC CPU2000. This program has a small memory footprint — less than 5MB total, but acquires this memory with many thousands of small allocations.

All SPEC2000 benchmarks, are provided with *test*, *train*, and *ref* datasets. For each dataset, output files are provided to for the purpose of validating results. For, `179.art`, the ref dataset consists of two parts, designated *ref.1* and *ref.2*.

The ref.2 dataset is used for comparison of `179.art` performance as it was found to be sufficient to reflect the trends in performance differences between various cache and hoard configurations.

7.1.5 `181.mcf` — min cost flow solver, transport scheduling

An integer benchmark from SPEC CPU2000. This program performs many indirect memory accesses while traversing a graph, yielding a much less cache-friendly program.

The ref dataset is used to test this program as the is enough work done to provide a clear difference between different methods.

7.1.6 `183.equake` — earthquake propagation simulation

A double-precision floating point benchmark from SPEC CPU2000. The core part of the computation is matrix multiplication.

The train dataset is used to test this program, as it was found that a single run of the ref dataset would exceed two hours in particularly unsuitable cache and hoard configurations. The train dataset completes in more reasonable times while yielding similar performance differences between hoard and cache configurations.

7.1.7 `mpeg2decode` — MPEG2 video decoder

The `mpeg2decode` program is an MPEG video decoder from the MediaBench suite. Running this program on the SPE gives an indication of the effort and performance results porting a simple media processing program to the SPE.

A 26 second, 320 $\times$ 240 MPEG1 video is used to test this program, providing an indicative baseline of performance.

7.2 Unsuccessful — `256.bzip2`

Another program from the SPEC CPU2000 suite, `256.bzip2` (which performs compression and decompression of various data), was pursued for testing use with the hoard. This program has the largest text size of any of those tested, some large static arrays and additionally makes heavy use of the stack. A number of methods were attempted to attempt to reduce the memory footprint, including the removal of as many functions and global variables as possible, and changing two large arrays to be allocated and accessed via the hoard.

The hoard itself attempts to dynamically allocate as much space as is possible for its storage, leaving a small amount of available free memory for the program stack. The de-

fault used for other programs is insufficient for `256.bzip2`, causing crashes as the program stack overwrote hoard storage. The program could be made to run correctly by configuring the hoard to allocate less space for page storage, but this further decreased the number of pages available for storing program data. Initial testing of this program showed very poor performance when using the SDK cache or hoard when compared to the PPE.

Because of the high degree of intrusion necessary, the complexity of this case and time limitations, the performance of `256.bzip2` will not be considered in this thesis.

7.3 Benchmark compilation

Each benchmark, apart from `julia_set`, is compiled as a standalone SPU program. The compiler used is the FSF release of GCC version 4.4.1.

Individual hoard and IBM SDK programs are compiled using the following compiler options:

`-ffast-math` As the SPE implements only a subset of the IEEE floating point standard, this option prevents extra code being generated to handle special cases that are not relevant to the benchmarks being run.

`-ffunction-sections` and **`-fdata-sections`** Instructs the compiler to place each function and global variable into its own section, allowing the linker to remove any that are unused, minimising text size.

`-Wl,--gc-sections` Instructs the linker to garbage collect unused sections from the program.

`-fno-exceptions` and **`-fno-rtti`** Ensures that the compiler does not attempt to generate exception or runtime-type information as it is not needed

`-O3` Seek to optimise the program heavily.

In addition to the above options, the `qsort` benchmark is compiled with

```
--param max-inline-insns-single=32000
--param large-function-growth=8000
--param inline-unit-growth=6000
--param max-inline-insns-auto=10000
```

This increases the degree of function-inlining performed and offers a large performance boost for that benchmark. These extra parameters were taken from the `qsort` makefile in the IBM SDK.

7.4 Benchmark modification

Each of the above benchmarks required some amount of modification to be able to make use of the hoard, IBM SDK Cache, and on the PPE. Chapter 6 explains the general design and implementation of the hoard user interface, and more specific descriptions of these changes are in the following sections of this chapter.

Appendix B contains a summary of the number of lines modified in each source file of the programs tested.

7.4.1 Language-related changes

The benchmarks were all written in C. To make use of hoard-based memory management, the program must be compiled as C++. The following classes of changes were necessary:

- Replace C++ keywords with non-keyword tokens. e.g. `class/class_`, `new/new_`, etc.
- Make pointer casts explicit, due to stronger typing in C++ e.g.

  ```
  int* ii = (int*)malloc(sizeof(int)*x);
  ```

- Convert K&R style function prototypes to C++ style. The protoize program (distributed with gcc) was used to covert between the two, where necessary e.g.

  ```
  void Add_Block(comp, bx, by, dct_type, addflag)
  int comp, bx, by, dct_type, addflag;
  ```

  becomes

  ```
  void Add_Block(int comp, int bx, int by, int dct_type, int addflag);
  ```

- Add explicit casts or temporary variables for vararg function calls `printf()` and `fscanf()`.

7.4.2 Abstraction-related changes

The declarations of any data that are to be managed by the hoard need to be changed from regular pointers to `hoard_ptr` equivalents (or typedefs thereof). The places where

this is necessary are made apparent as compiler warnings are generated anywhere that a `hoard_ptr` is being assigned to a regular pointer.

In particular, the hoard-provided `malloc()` family of functions return `hoard_ptr<void>` typed data, which will cause an error to be generated anywhere that the result from a call to `malloc()` is being assigned to a regular pointer.

It is a straightforward (and generally mundane) process to work through the program and change declared types from regular pointer types to hoard-managed types.

`julia_set` is written in a way that is closely tied to the IBM SDK cache — particularly using a function `cache_rd_x4()` to load four texels at once into a vector. An analogous `hoard_rd_x4()` function was implemented to load data in a similar fashion for the hoard.

The method of allocating (and managing) pointers is different between the IBM cache and hoard — the IBM cache uses regular system addresses (effective address, or ea), where the hoard uses its hoard address form for performing data lookup.

For `julia_set`, unlike the other benchmarks, memory allocation is performed by the PPE, not the SPE. The effective addresses of texture data that are passed to the SPU context must be converted and processed at the start of the SPE program's execution. This is achieved using the hoard function `get_hoard_pointer()`.

The existing hoard implementation does not support the freeing of allocated memory. None of the benchmark programs used has a need to be able to `free()` allocated data, and it is not expected to be commonly needed for SPU programs.

So that there is no difference attributable to allocator overhead, the IBM cache, hoard and PPE programs are all linked with a pooling memory allocator without support for `free()`.

7.4.3 Changes related to building comparable versions

These changes allow the same file(s) to be compiled with multiple configurations — hoard, IBM cache, and to run on the PPE. These are necessary due to the differences in datatypes and intrinsics between platforms (e.g. different SIMD intrinsics for SPE and PPE, see IBM Corporation et al. (2008a)), as well as alternate implementations of some system headers (e.g. `hoard/mman.h` replacing `sys/mman.h`).

Changes of this kind were made to `qsort` and `hsort`.

In addition, `qsort` and `hsort` were modified to be deterministic across architectures

— particularly to ensure no difference for pseudo-random number generation. Datasets are loaded from disk where necessary to maintain consistency. Times measured are for dataset sorting only.

For these benchmarks, pseudo-random program execution steps were replaced with non-random alternatives, e.g. in `qsort`, the first element is used as the pivot rather than a random element.

7.4.4 Limited RAM on PlayStation 3

`julia_set` is written to use 'huge' pages (16MB per page) to minimise any overhead for TLB misses when accessing input and output data. The program as distributed in the SDK attempts to allocate a total of eight huge pages — eight 16MB blocks of RAM. It was found to be difficult to obtain more than five or six 16MB pages on the PlayStation 3 with a running Linux system. Consequently, the program was modified to more efficiently store texture data and framebuffers, and so require fewer of these huge pages.

7.4.5 Shadow reference specialisations

The `179.art` and `181.mcf` benchmarks required explicit declaration of shadow template types as described in 6.2.2.1.

7.5 Experimentation

To test the hoard, and to compare it to other caching methods, each of the above benchmark was run in the following configurations:

1. As a native PPE program.
2. Using the IBM SDK cache on a SPE, with each available cache line size, from 16B to 16KB.
3. Using the hoard on a SPE, with each available page size, from 1KB to 16KB.

7.5.1 Rationale

Running the program on the PPE provides a indicative speed baseline for comparison with other programs. PPE programs are compiled with the same version of GCC, make use of

the same system memory architecture (albeit with memory operations controlled differently) and all cores have the same clock speed. The differences between the PPE and SPE architectures should be taken into account when comparing the runtimes of programs on each, but the speed differences help to assess the real-world usefulness of a caching system.

Testing multiple page sizes provides a clear indication of the relationship between the page size and performance.

Based on the results of these tests, it is possible to compare the various methods for performance in different configurations, to make a judgement of the general and specific usefulness of software-managed caches for the Cell BE SPE, and to identify shortcomings in each approach, particularly the hoard, that might indicate opportunities for improvement.

As has been stated elsewhere, the source for COMIC became available too late in this research to allow direct comparison of performance with the hoard. Section 8.9 does give some broad consideration to the differences between the two systems and considers probable performance outcomes in response to the results observed through the testing of the hoard and SDK cache.

There is no comparison with IBM's XL compiler in this research. The compiler performs analysis of memory accesses in program code and generates code appropriate for each using a separate data buffer and software cache. Like COMIC, it has a much broader scope than the hoard, but was also unavailable at the time the research was performed.

7.5.2 Measurement confidence

The length of each program run was measured using the GNU time program (Free Software Foundation 1999) which provides the wall-clock time of the complete program run, with millisecond accuracy. The one exception to this is `julia_set` which has an internal timer that measures the total time to render all frames.

The results from individual program runs are highly repeatable with observed variation in runtime between runs being less than 0.05% for most benchmarks. The variation that is observed appears to be attributable to disk latency and operating system overheads. While the programs are running on an SPE, they are never de-scheduled, and so run uninterrupted until completion. For `qsort`, an observed variation of up to 10ms is a larger percentage of the runtime — up to 0.2%.

Because of the low degree of variation in program runtimes, most program configura-

tions were tested once only. For any case where there was a result that did not appear to match expected or otherwise observed results, the program was re-run to verify. In each case, the repeated program run matched the original result.

This low variance indicates a high degree of accuracy in the measurements recorded. This level of accuracy permits accurate comparison of results with difference configurations, providing a high degree of confidence when comparing the performance benefits of one method over another. For the purpose of assessing the merit of on configuration over another, differences of greater than 10% are considered to be of particular interest.

Chapter 8

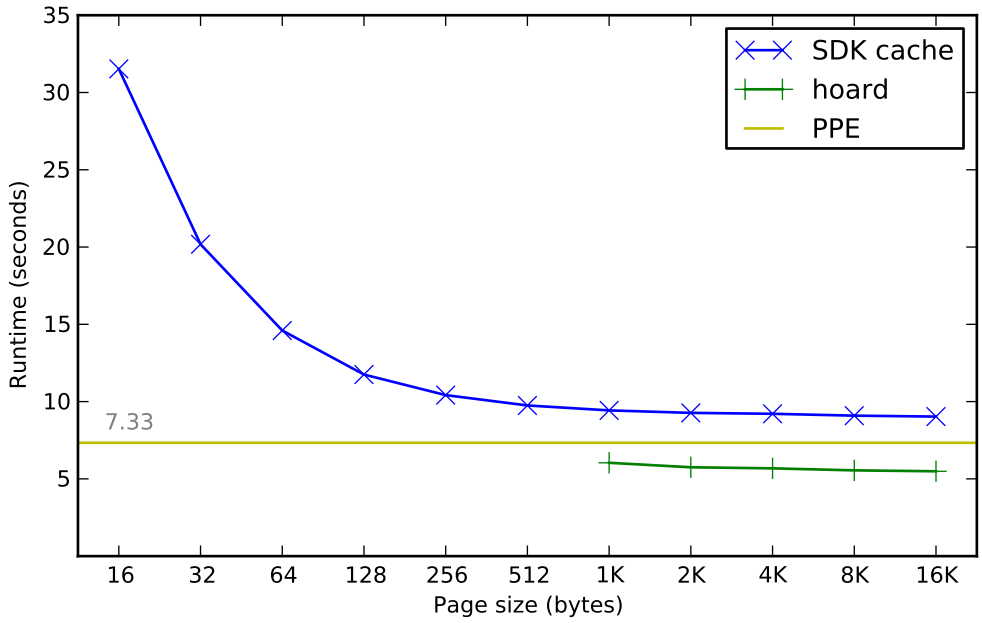
Results & Evaluation

In this chapter, the performance of each of the selected benchmarks will be presented, along with an analysis of the results. Some comparison will be made between the various configurations and potential opportunities will be identified for improving the performance of the hoard implementation. Implementation and testing of the potential improvements will be considered in the next chapter.

8.1 `qsort`

As mentioned earlier, quicksort is a sorting algorithm that is a high-performing general-purpose sorting algorithm which has quite good locality of access when compared to many other sorting algorithms. The `qsort` program is distributed with the IBM Cell SDK to demonstrate the use of the SDK software cache that performs a quicksort on an array of single-precision floating point numbers, each stored in a 16 byte vector.

The graph in Figure 8.1 shows the runtime performance for the `qsort` program for a data-set consisting of 4,194,304 (2^{22}) 16 byte items when run with each page size for the hoard, SDK cache, and PPE. The yellow line on the graph indicates the runtime of the same program and data-set when run on the PPE, without any software caching or explicit extra memory management (for the PPE performance line, the page size on the X-axis is irrelevant — the line crosses the whole graph to make the comparison clear). This point of comparison provides an indication of the usefulness of using an SPE with a general memory manager.



Page size	16	32	64	128	256	512	1K	2K	4K	8K	16K
SDK cache	31.53	20.18	14.60	11.75	10.42	9.75	9.43	9.27	9.21	9.09	9.03
hoard							6.04	5.75	5.68	5.55	5.49

Figure 8.1: Runtimes of hoard and SDK cache for `qsort`

The blue line shows the IBM SDK cache runtimes, for block sizes from sixteen bytes to sixteen kilobytes, and the green line shows the hoard runtimes for cache sizes from one to sixteen kilobytes — all available size configurations for both system. The runtimes for each point are presented in the table underneath each graph for easy reference. The ideal configuration is the one with the shortest runtime.

It is clear that it is faster to run the sort on an SPE using the hoard, where it is able to complete the process in ~70% of the time taken by the PPE, and ~60% of the time of the SDK cache. The runtime improves for both memory managers as the page size increases, reflecting the spacial-local access patterns of this quicksort implementation coupled with the higher throughput of larger DMA transfers.

The large performance decrease for the SDK cache for smaller page sizes stands out, and the cause becomes clearer when recalling some details of the SPE's MFC and examining the DMA statistics for each case.

Figure 8.2 depicts the graph of the number of DMA operations (gets and puts) issued by the `qsort` program throughout the program run. For this program, every page that is loaded to the SPU is modified and is written back to memory.

Smaller page sizes result in more DMA operations than larger pages, again attributable to the typical space-local memory accesses of the quicksort algorithm. The SPE MFC transfers whole 128 byte memory lines. The consequence of this is that for pages smaller than 128 bytes, misses are more frequent but the time spent waiting for the completion of a memory request does not reduce, resulting in a time delay that increases beyond what might otherwise be expected, yielding the steep increase in runtime that may be observed in Figure 8.1.

Figure 8.3 shows the total amount of data transferred by `qsort` programs throughout a complete run (disregarding the architectural overhead involved in sub-128 byte transfers). The hoard transfers less data overall in most configurations, with only a small increase with page size. The SDK cache shows a large increase in overall data transferred as page size increases. This is attributable to the set-associativity and size of the SDK cache: as page size increases, the number of sets decreases resulting in more conflict misses.

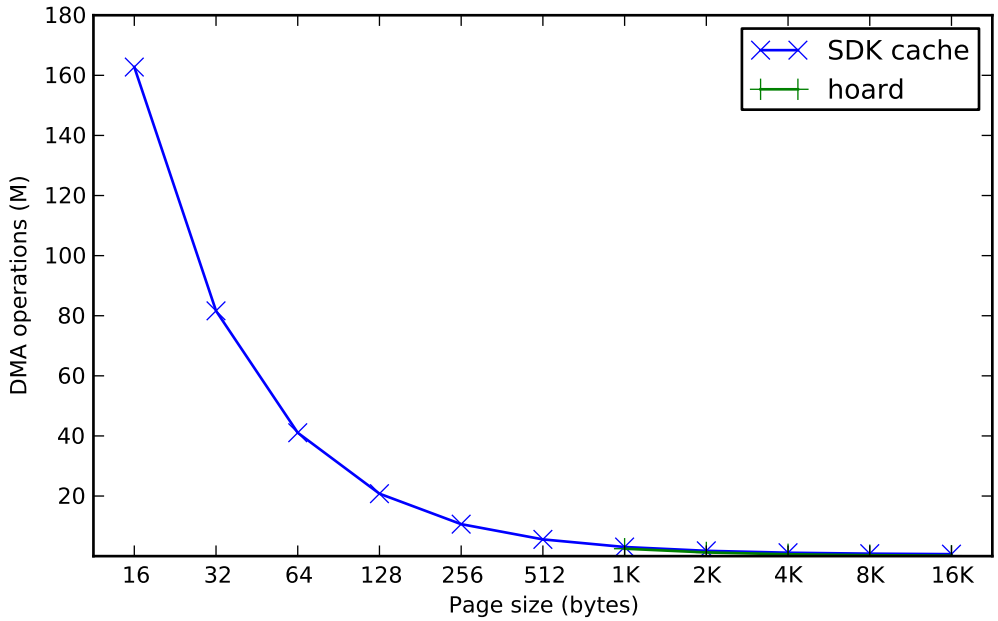
In addition to this, the SDK cache always has a total of 128KB of storage space, whereas the hoard is able to use all storage that is available. For `qsort`, this varies between 176KB and 198KB depending upon the page size[1]. As such, there will be more capacity misses for the SDK cache.

It may seem contradictory that as the amount of data transferred increases, the runtime of both memory managers decreases. It is worth considering the amount of data transferred against the physical limits of the architecture.

The red line in the first graph in Figure 8.4 shows a measured maximum throughput of data for a program performing DMA gets and puts in the pattern used by the hoard or SDK cache — gets fenced after puts[2]. The second graph shows the same numbers as a percentage of peak throughput. It is clear from theses graphs that the amount of DMA traffic for large page sizes is very small and the miss rate of the SDK cache decreases. Additionally, the cause of the large runtime of the SDK cache when using small pages becomes clearer as the program is very much bound by the low DMA throughput for this configuration.

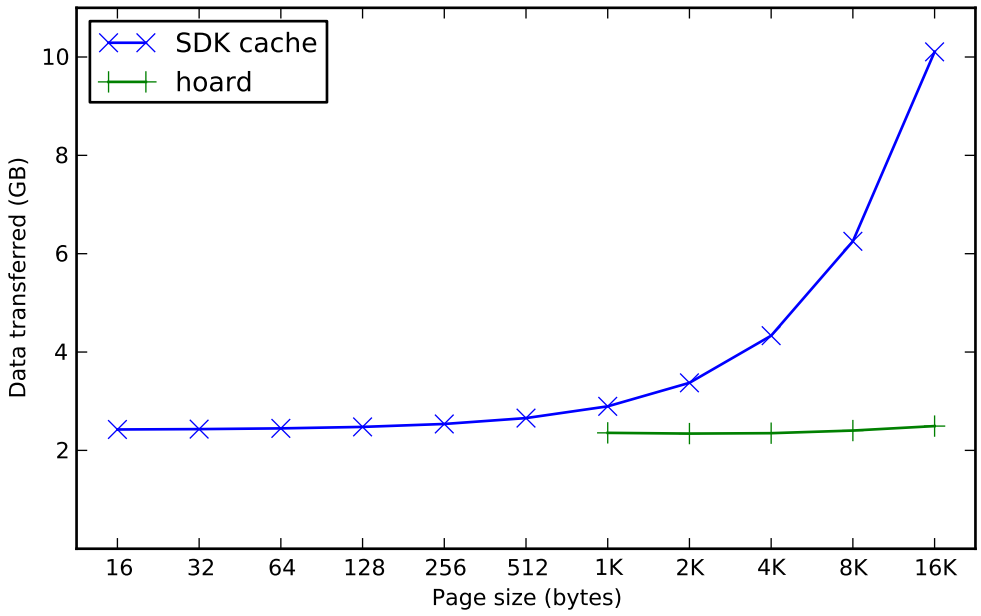
[1] This variation is a consequence of the natural alignment of pages used.

[2] Numbers were generated using a modified version of the dmabench program from the IBM SDK for multicore acceleration 3.1



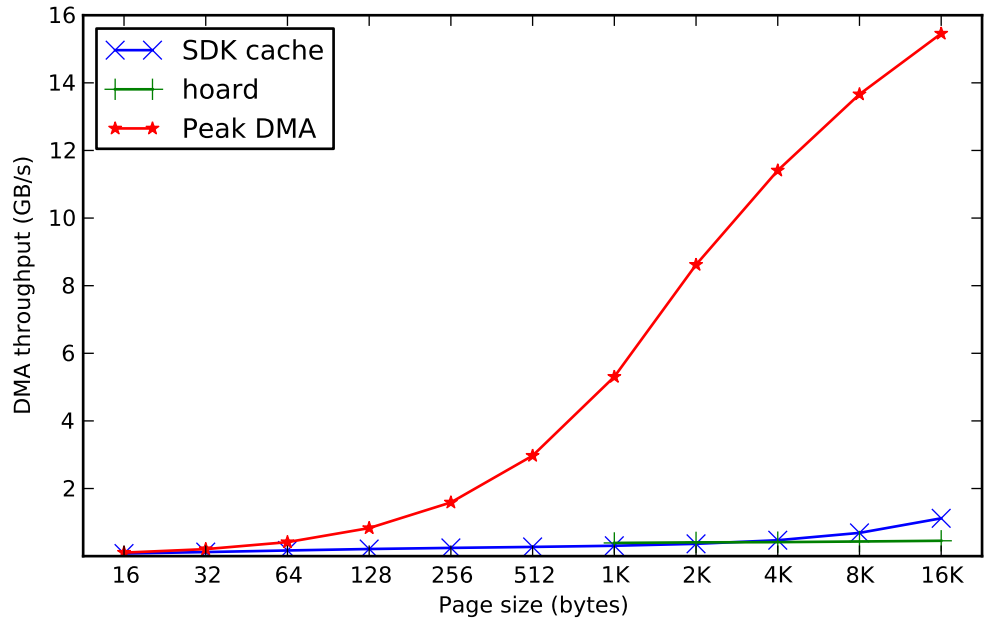
Page size	16	32	64	128	256	512	1K	2K	4K	8K	16K
SDK cache	162.75	81.63	41.06	20.78	10.64	5.57	3.04	1.77	1.14	0.82	0.66
hoard							2.47	1.23	0.62	0.32	0.16

Figure 8.2: DMA operation count of hoard and SDK cache for `qsort`

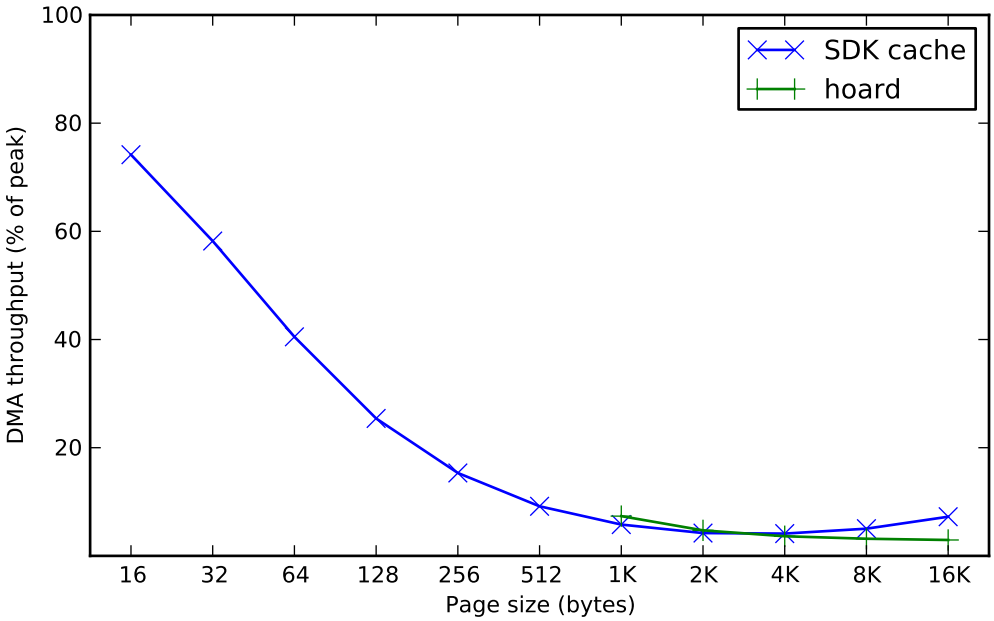


Page size	16	32	64	128	256	512	1K	2K	4K	8K	16K
SDK cache	2.43	2.43	2.45	2.48	2.54	2.66	2.90	3.37	4.33	6.25	10.10
hoard							2.36	2.34	2.35	2.40	2.49

Figure 8.3: Total data transfer of hoard and SDK cache for `qsort`



(a) Total average DMA throughput



(b) Average DMA throughput as a percentage of peak

Page size	16	32	64	128	256	512	1K	2K	4K	8K	16K
Peak	0.10	0.21	0.41	0.83	1.59	2.97	5.31	8.62	11.41	13.66	15.46
SDK cache	0.08	0.12	0.17	0.21	0.24	0.27	0.31	0.36	0.47	0.69	1.12
hoard							0.39	0.41	0.41	0.43	0.45
SDK cache %	74.17	58.21	40.5	25.41	15.33	9.17	5.79	4.22	4.12	5.03	7.24
hoard %							7.35	4.72	3.63	3.17	2.94

Figure 8.4: Average DMA throughput of hoard and SDK cache for `qsort`

8.1.1 Summary for `qsort`

These results indicate that the increased efficiency of larger DMA operations and the higher hit rate for the larger, fully associative hoard storage result in excellent performance for `qsort` when run with the hoard using large pages, and that this simple program shows a clear benefit when run on the SPE instead of the PPE.

The relatively high data locality and the difference between actual DMA throughput and the peak rate indicate that there may be some performance to be gained by performing extra data transfers asynchronously, such as pre-fetching data, but that any improvement is unlikely to make a large difference to the runtime of this program.

8.2 `hsort`

Heap sort exhibits very different memory access characteristics to those of quicksort. This can be seen particularly in the `heapdown()` function in `hsort.c`:

```
void heapdown (item_tp a, int start, int count) {
    int root, child;
    for (root = start; child = (root * 2 + 1), child < count;) {
        item_t child_val = a[child];
        item_t chpp_val = a[child+1];
        if (child < (count - 1) &&
            compare_leq (child_val, chpp_val)) {
            child += 1;
            child_val = chpp_val;
        }
        if (compare_and_swap (a, root, child, child_val, 0))
            root = child;
        else
            return;
    }
}
```

This function is called once for each item in the array. In it, the loop action of `child = (root * 2 + 1)` results in the data-set being stepped through in such a way that up to

$\log_2 n$ items from the array are accessed, progressively further and further apart. Simplistically, if $\log_2 n > 2B$ (where B is the number of pages available in the cache) then it can expected that in the worst case the cache will be flushed in each iteration of the `for` loop in `heapdown()`. As such, the overall performance can be expected to decrease if the number of available pages decreases below the working set of pages needed for a particular data-set size.

Figure 8.5 shows a very different graph shape to that for `qsort`. The SPE programs run slower than the PPE equivalent. For small page sizes and the SDK cache there is a clear slow-down, although this is much less than was the case for `qsort`. The fastest configuration for the SDK cache is 512 byte pages, suggesting that performance of the hoard might be able to be improved if page size can be reduced — and storage efficiency increased. For larger page sizes, both methods show poorer performance. The fastest hoard configuration is 6.7% slower than the fastest SDK cache configuration, which is undesirable, but within a reasonable margin.

The data point for the hoard with 1KB sizes is interesting — it defies the trend of the other points on the hoard line, and the trend of the relationship between page size and runtime for the SDK cache. In this case, the 4KB size of the d-pages and the low spacial locality of memory accesses result in a large reduction of the number of pages available for regular data storage, by up to a factor of five, resulting in the poorer performance of this configuration when compared to the larger page size.

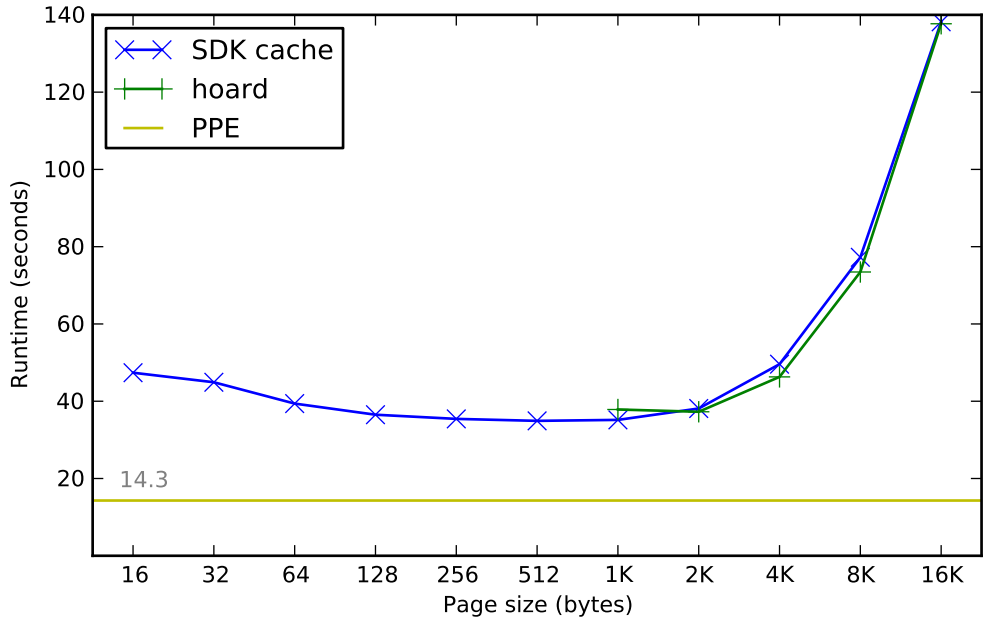
Figure 8.6 shows that the number of DMA requests does decrease as the page size increases, but unlike `qsort` this does not continue to the largest page sizes. When the number of pages is smaller than the number needed for the working set, the number of requests starts increasing. The high number of DMA requests for hoard page sizes of 1 and 2KB relate to the low-locality page-table-size effects mentioned above.

Where the hoard writes every page back to memory after use, the SDK cache tracks whether data has been modified and writes back only the pages that have changed. Unlike `qsort`, `hsort` shows a divergence between page fetches and write-backs. Figure 8.7 shows the same data as Figure 8.6, separated into lines for DMA *get*s and *put*s. There is a wide gap between the cache fetches and writes for small pages, but as the page size increases, the gap becomes negligible.

The hit rate, as shown in Figure 8.8, shows similar information to the previous two graphs, expressing the percentage of DMA operations that do not result in a DMA transfer. It can be seen that in the best case both memory managers have hit rates of around 92%. The hit rate decreases at either extreme, by around 10% for the smallest pages, and by 1.4% for the largest pages.

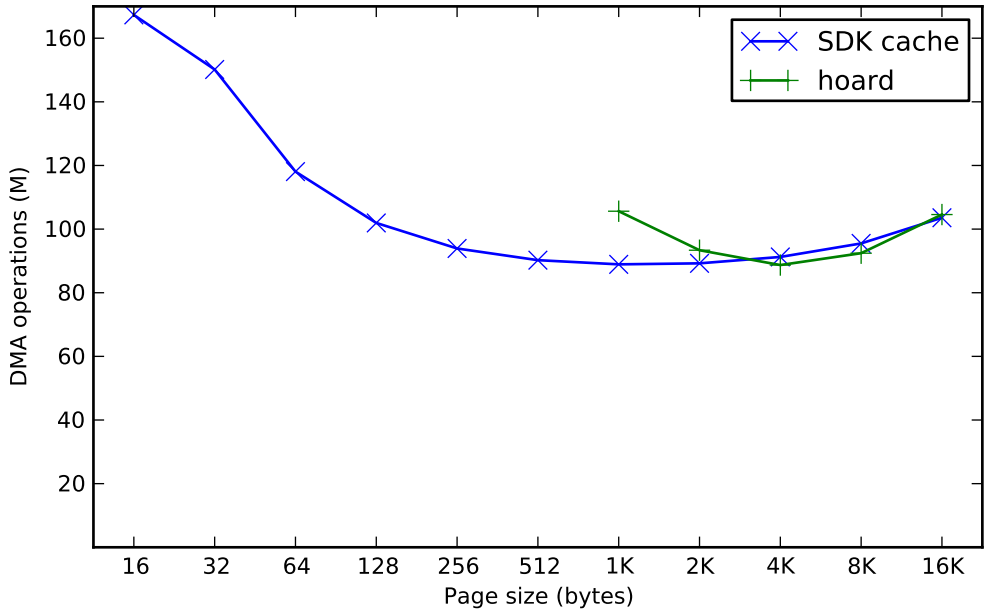
Figure 8.9 shows that the decrease in hit rate for the largest problems results in a huge increase in overall DMA traffic, and Figure 8.10 indicates that the DMA throughput of each program is approaching the limits of the architecture. It is clear that regardless of the configuration, a large portion of the program's execution time is being spent transferring data.

The 4 and 8KB page configurations exhibit the interesting characteristic that they have lower runtimes, higher hit rates (resulting in lower total data transferred), but overall higher data throughput.



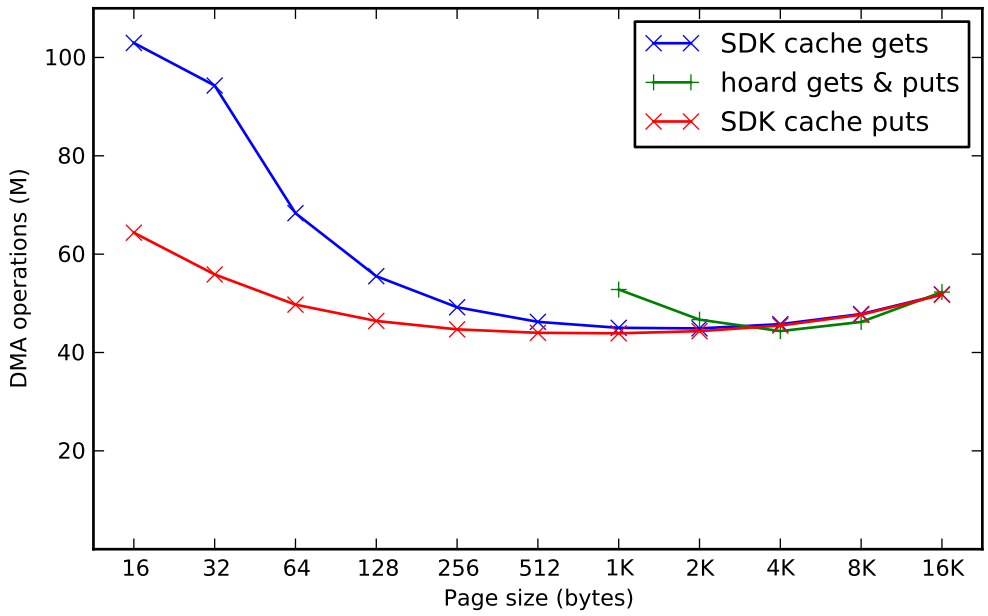
Page size	16	32	64	128	256	512	1K	2K	4K	8K	16K
SDK cache	47.38	44.91	39.39	36.51	35.43	34.92	35.17	38.12	49.59	77.25	138.20
hoard							37.86	37.27	46.30	73.45	137.70

Figure 8.5: Runtimes of hoard and SDK cache for `hsort`



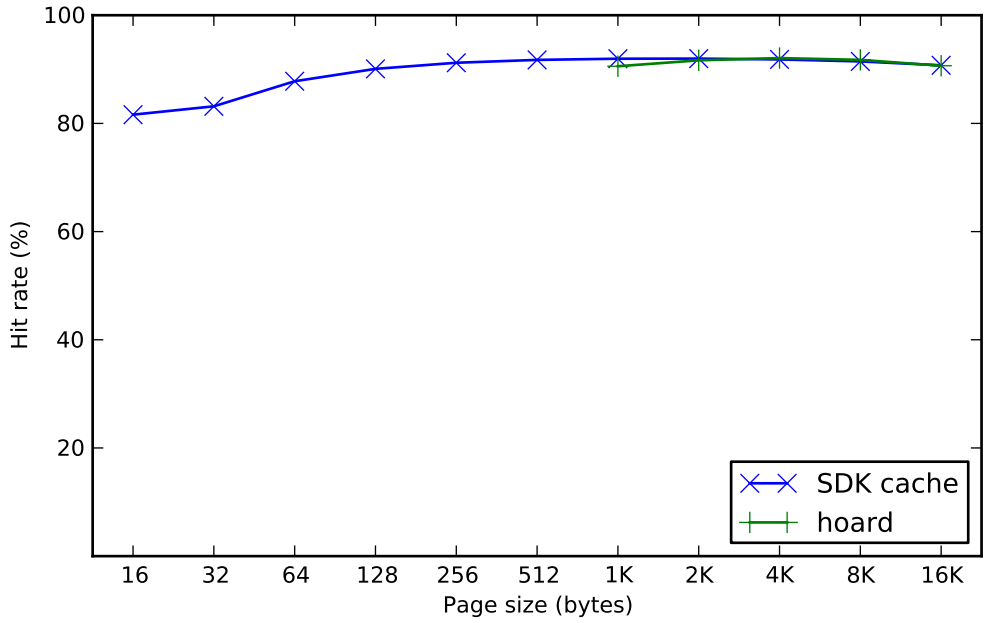
Page size	16	32	64	128	256	512	1K	2K	4K	8K	16K
SDK cache	167.32	150.16	118.07	101.94	93.91	90.24	88.92	89.24	91.25	95.50	103.57
hoard							105.62	93.36	88.70	92.43	104.57

Figure 8.6: DMA operation count of hoard and SDK cache for `hsort`



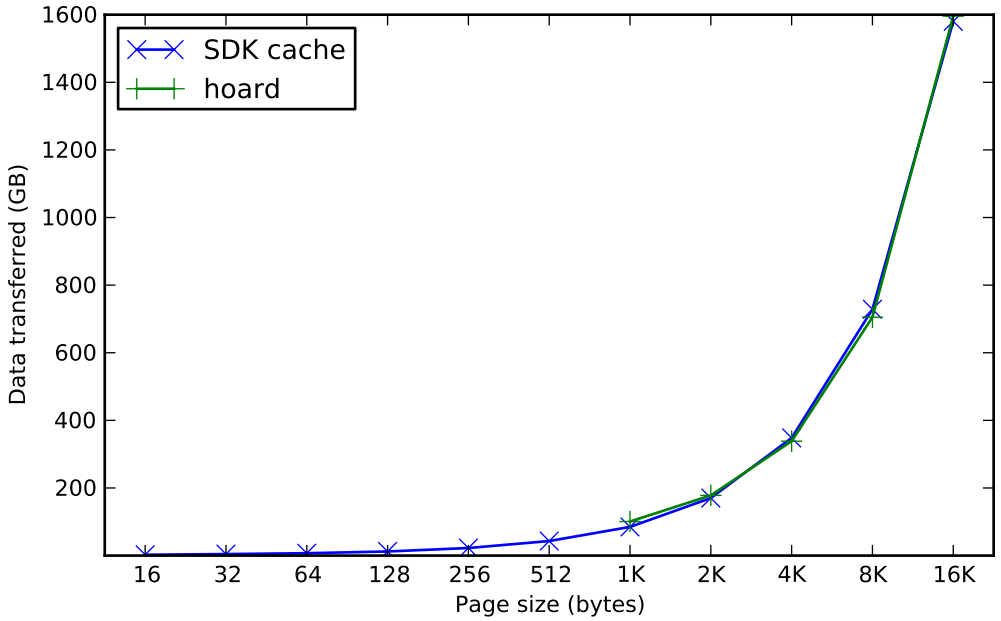
Page size	16	32	64	128	256	512	1K	2K	4K	8K	16K
SDK cache gets	102.95	94.29	68.34	55.52	49.20	46.23	45.02	44.90	45.76	47.82	51.82
hoard gets & puts							52.81	46.68	44.35	46.21	52.29
SDK cache puts	64.37	55.87	49.73	46.42	44.71	44.00	43.90	44.34	45.48	47.68	51.75

Figure 8.7: Split DMA *get* and *put* counts of hoard and SDK cache for `hsort`



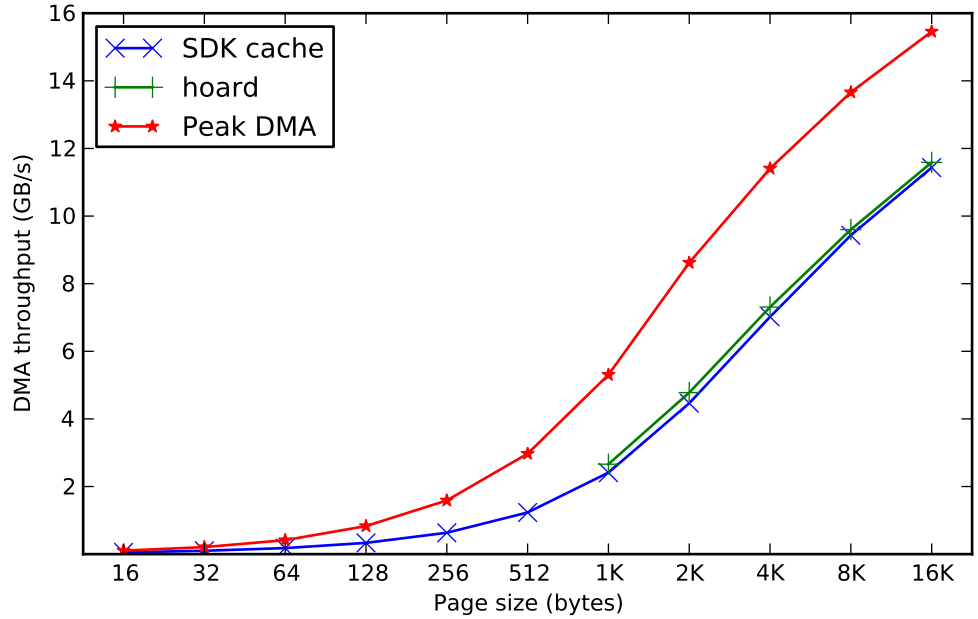
Page size	16	32	64	128	256	512	1K	2K	4K	8K	16K
SDK cache	81.61	83.16	87.80	90.09	91.21	91.74	91.96	91.98	91.83	91.46	90.75
hoard							90.57	91.66	92.08	91.75	90.66

Figure 8.8: Hit rates of hoard and SDK cache for `hsort`

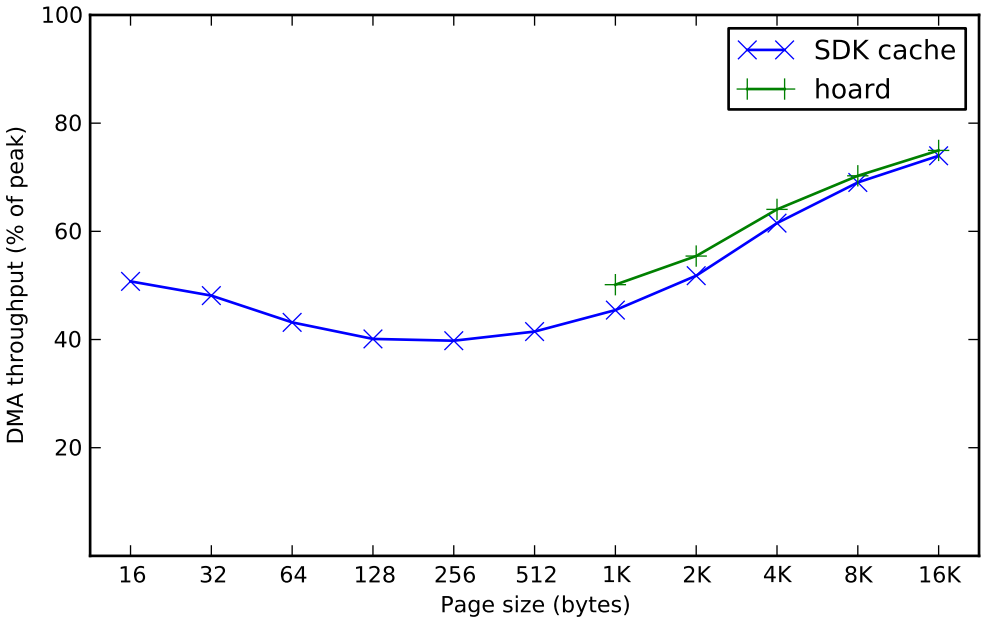


Page size	16	32	64	128	256	512	1K	2K	4K	8K	16K
SDK cache	2.49	4.48	7.04	12.15	22.39	43.03	84.80	170.2	348.1	728.6	1,580
hoard							100.7	178.1	338.4	705.2	1,595

Figure 8.9: Total data transfer of hoard and SDK cache for `hsort`



(a) Total average DMA throughput



(b) Average DMA throughput as a percentage of peak

Page size	16	32	64	128	256	512	1K	2K	4K	8K	16K
Peak	0.10	0.21	0.41	0.83	1.59	2.97	5.31	8.62	11.41	13.66	15.46
SDK cache	0.05	0.10	0.18	0.33	0.63	1.23	2.41	4.47	7.02	9.43	11.43
hoard							2.66	4.78	7.31	9.60	11.59
SDK cache %	0.51	0.48	0.43	0.4	0.4	0.41	0.45	0.52	0.62	0.69	0.74
hoard %							0.5	0.55	0.64	0.7	0.75

Figure 8.10: Average DMA throughput of hoard and SDK cache for `hsort`

8.2.1 Summary for `hsort`

The total amount of data to be transferred is clearly a limiting factor in this benchmark, as shown by the high levels of DMA utilisation in Figure 8.10. This results in the SPE programs running — at best — almost 2.5 times slower than the PPE version.

To help improve these results, it may be beneficial to attempt to reduce the amount of time spent waiting for DMA traffic to complete. Ways of doing this may include keeping one or more pages free to reduce the total time penalty of a miss, writing back only modified pages or fetching partial page sections. It may also be possible to use a more accurate page-replacement algorithm to reduce the total number of misses. Possible improvements to the hoard are examined in Chapter 9.

Additionally, there may be some benefit in changing the relative sizes of the hoard lookup tables when using small page sizes to increase the number of usable hoard pages.

8.3 `julia_set`

The `julia_set` program is unique among the benchmarks used in that it is not a standalone application that runs on a single core, SPE, or PPE. It is written to use the PPE as a job scheduler and whichever SPEs are available. It renders a series of frames of a time-varying three-dimensional Julia set fractal, and dynamically adjusts the amount of work done by each SPE from one frame to the next in an attempt to minimise total frame generation time.

The following results were generated by rendering 100 frames using all six available SPEs. The default configuration of this program was used as provided with the IBM SDK for Multicore Acceleration 3.1.

It can be seen from the runtime comparison in Figure 8.13 that the hoard-based version

Figure 8.11: Images rendered with `julia_set`

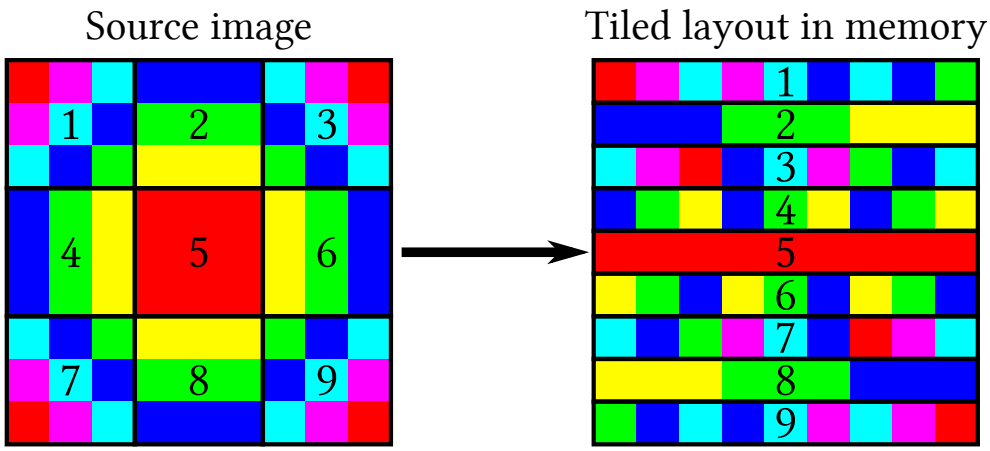
runs 10–11% faster than the SDK cache version regardless of page size.

In rendering the scene, `julia_set` uses the memory manager to access texture data. Textures are stored in 16×16 pixel (512 byte) tiles to increase the likelihood of a cache hit. Tiles are stored aligned in memory, the result being that for page sizes of 512 bytes or larger, whole texture tiles are loaded into local store as needed.

Figure 8.12 illustrates texture tiling. Tiling is used to increase the rate of cache hits, and it can be seen that the hoard and SDK cache are both performing at their best when using block sizes that are large enough to hold one–four texture tiles — that is, with page sizes from 512B–2KB.

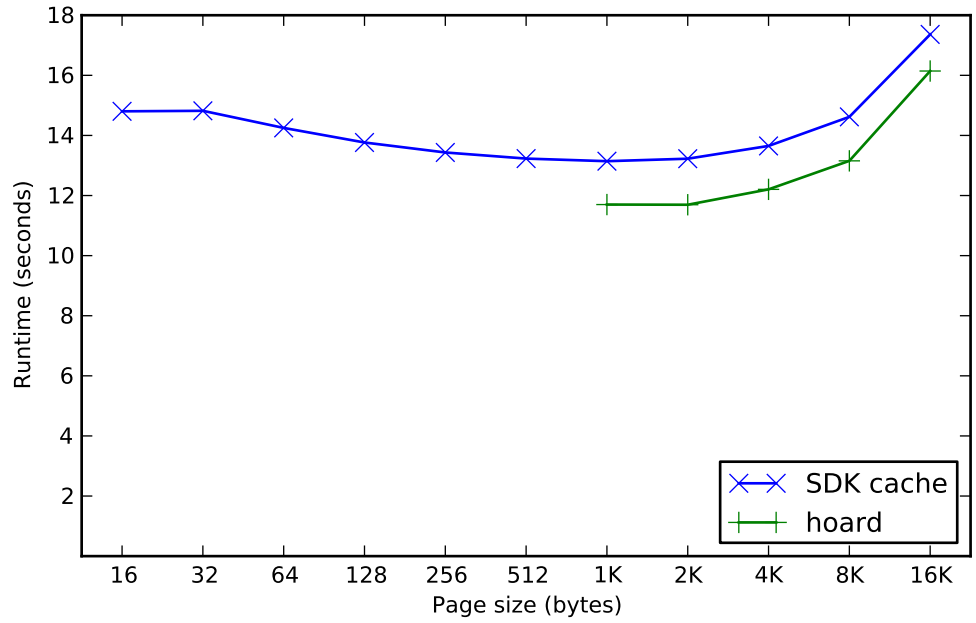
Figures 8.14 and 8.15 show that the hit rate drops rapidly as page sizes get smaller, which demonstrates that the tiling of the texture data is effective in increasing the cache performance for larger page sizes. For page sizes above 512 bytes, there is a slight decrease in hit rate, which can be explained by the fact that the tiling method used does not scale as effectively to larger page sizes — the texture data is arranged in tiles, but the tiles themselves are still linearly arranged, so the benefit of loading multiple tiles is limited.

The number of available pages (nine pages for the 16KB hoard configuration) is only three more than the number of texture lookups done per pixel (six), which will result in capacity misses when rendering more complex parts of the scene.



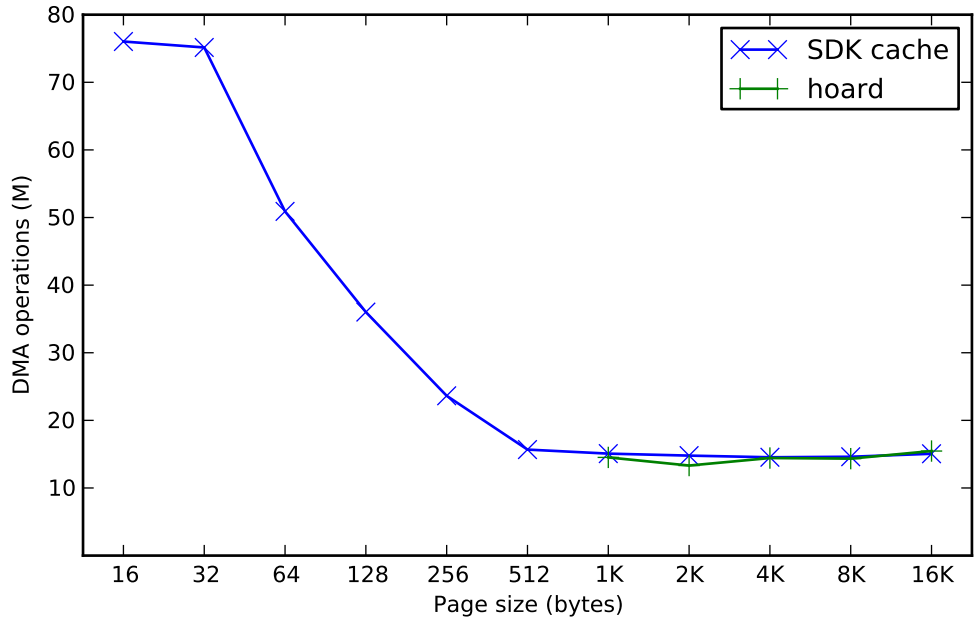
Tiled image storage: A source image is divided into a series of two dimensional tiles. Rather than being stored in memory as a sequential series of rows, the rows of each tile are stored sequentially. When access to textures occur with high two dimensional spacial locality, tiling can increases the effectiveness of a cache. When accessing the texture, the program must perform the necessary coordinate transformation to locate the correct texture data in the tiled storage.

Figure 8.12: Example of a tiled texture



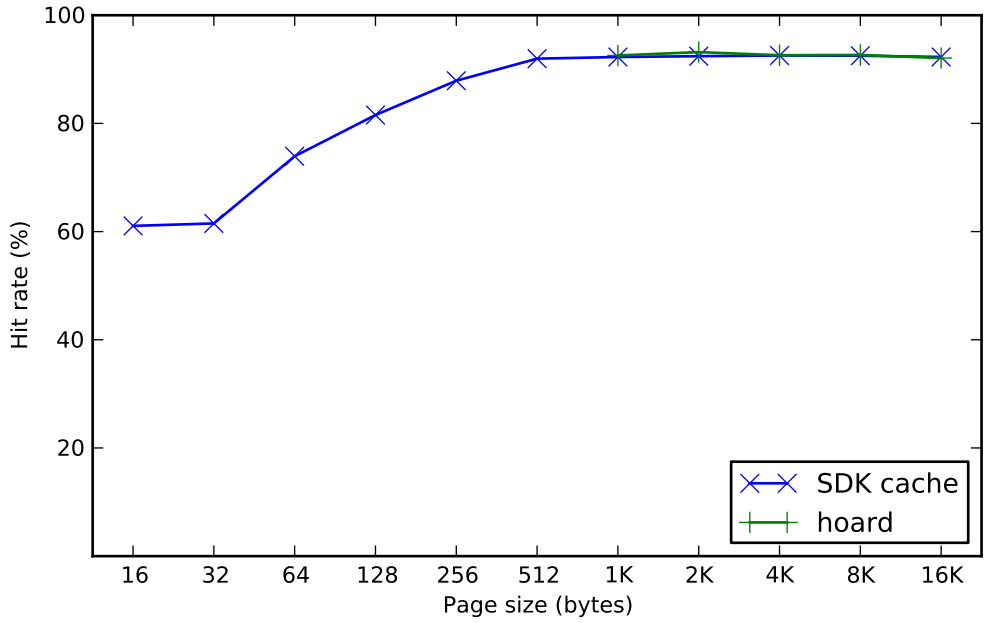
Page size	16	32	64	128	256	512	1K	2K	4K	8K	16K
SDK cache	14.80	14.82	14.25	13.77	13.43	13.23	13.14	13.23	13.65	14.62	17.36
hoard							11.70	11.69	12.21	13.15	16.14

Figure 8.13: Runtimes of hoard and SDK cache for `julia_set`



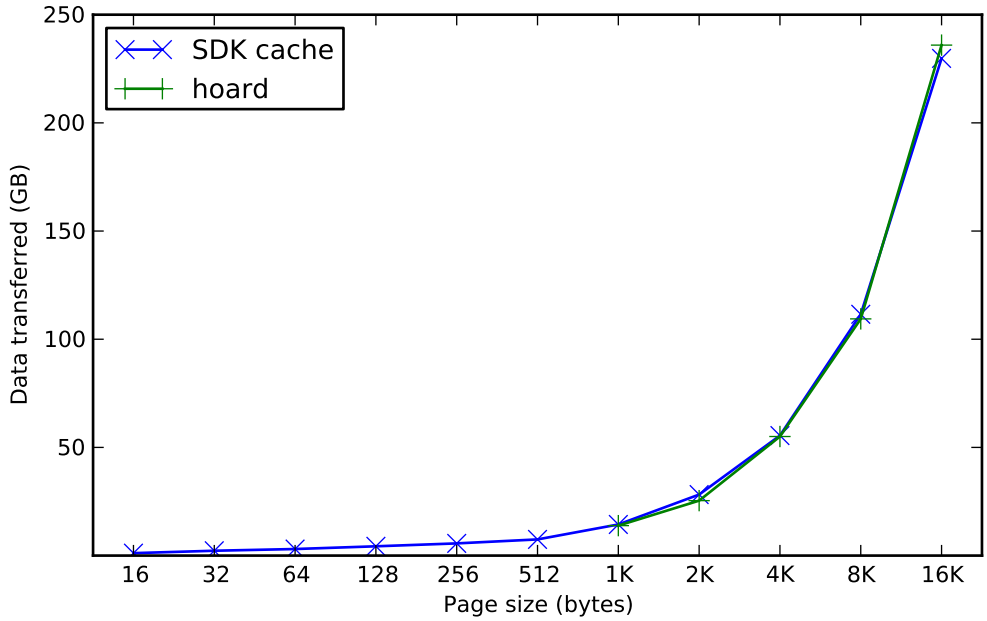
Page size	16	32	64	128	256	512	1K	2K	4K	8K	16K
SDK cache	76.03	75.15	50.90	36.02	23.64	15.68	15.08	14.79	14.54	14.62	15.06
hoard							14.52	13.32	14.43	14.34	15.46

Figure 8.14: DMA operation count of hoard and SDK cache for `julia_set`



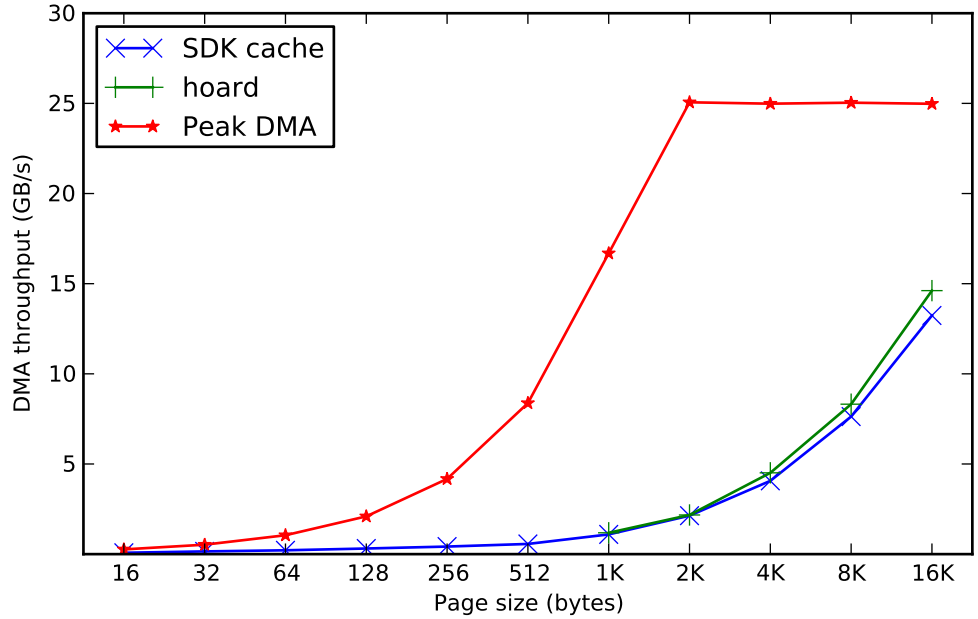
Page size	16	32	64	128	256	512	1K	2K	4K	8K	16K
SDK cache	61.06	61.51	73.93	81.55	87.89	91.97	92.28	92.43	92.55	92.51	92.29
hoard							92.56	93.18	92.61	92.66	92.08

Figure 8.15: Hit rates of hoard and SDK cache for `julia_set`

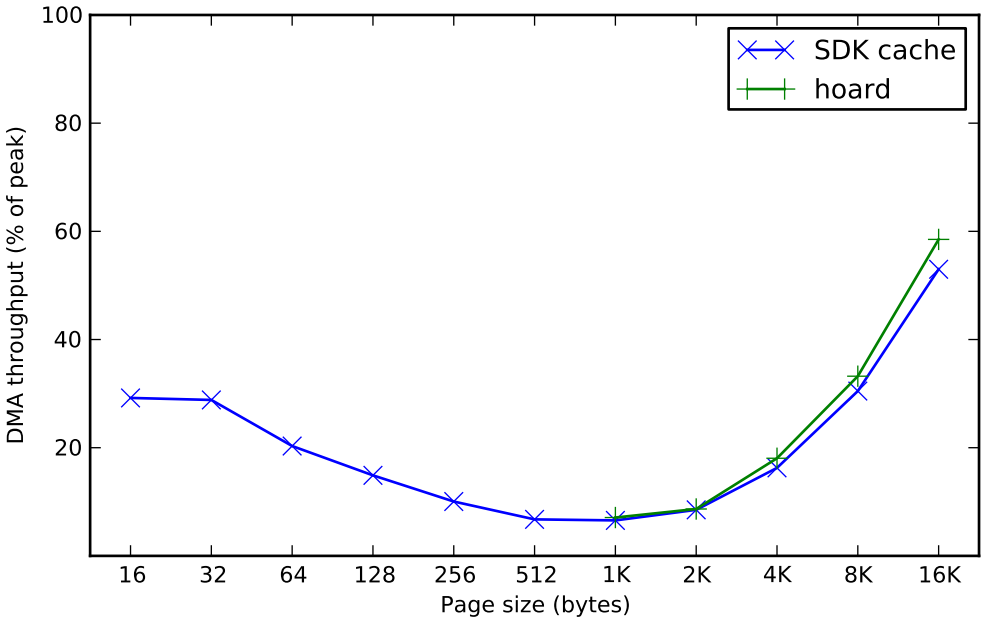


Page size	16	32	64	128	256	512	1K	2K	4K	8K	16K
SDK cache	1.13	2.24	3.03	4.29	5.64	7.48	14.38	28.21	55.48	111.53	229.80
hoard							13.85	25.40	55.04	109.40	235.94

Figure 8.16: Total data transfer of hoard and SDK cache for `julia_set`



(a) Total average DMA throughput



(b) Average DMA throughput as a percentage of peak

Page size	16	32	64	128	256	512	1K	2K	4K	8K	16K
Peak	0.26	0.52	1.05	2.10	4.17	8.38	16.68	25.06	24.98	25.04	24.98
SDK cache	0.08	0.15	0.21	0.31	0.42	0.57	1.09	2.13	4.06	7.63	13.24
hoard							1.18	2.17	4.51	8.32	14.62
SDK cache %	29.2	28.83	20.31	14.89	10.05	6.75	6.56	8.51	16.27	30.47	52.99
hoard %							7.09	8.67	18.05	33.22	58.51

Figure 8.17: Average DMA throughput of hoard and SDK cache for `julia_set`

Figure 8.16 has a similar shape to the corresponding graph for `hsort` (Figure 8.9), where a sharp rise in DMA traffic may be observed for larger page sizes. Figure 8.17 shows the rate of throughput for the SPE programs in comparison to the limit of the architecture for multiple SPEs, and it is clear that there is a much wider gap between the actual DMA throughput of the SPE programs compared to the limit — unlike `hsort`.

The hoard again exhibits higher hit rates and lower runtimes, resulting in an overall higher DMA data throughput.

8.3.1 Summary for `julia_set`

It may be possible to further improve the performance of `julia_set` through smarter pre-fetching of texture data, although tuning to the specific program may be necessary.

The performance for the smallest page size again does not seem to follow the trend of the larger sizes — there may be benefits available by changing the relative sizes of the hoard's page tables. Additionally, the total data size for the textures in this benchmark is approximately 32MB. Reducing the total address-space of the hoard will allow further optimisation of data lookup.

It would be of interest to run this program on a suitably configured Cell blade to see how the methods scale with more SPEs.

8.4 `179.art`

The workload of `179.art` involves a large amount of linear array traversal, each of which has a relatively small total memory footprint — in the order of two megabytes. The configurations tested for this program do not include the smallest page sizes for the SDK cache (16 or 32 bytes) as this program uses data structures with a size of 64 bytes, and the SDK cache does not support caching of data-types that are larger than a single cache line.

Two slightly different workloads are provided with `179.art` — ref.1 and ref.2. Both were tested and produce results that are similar enough that only ref.2 (the larger, longer-running, of the two) will be presented in this section.

It can be seen in Figure 8.18 that the runtime pattern is similar to that of `qsort`, in that the runtime decreases as block size increases (up to 8KB pages for both memory managers — performance decreases slightly for 16KB pages), and that the hoard version outperforms

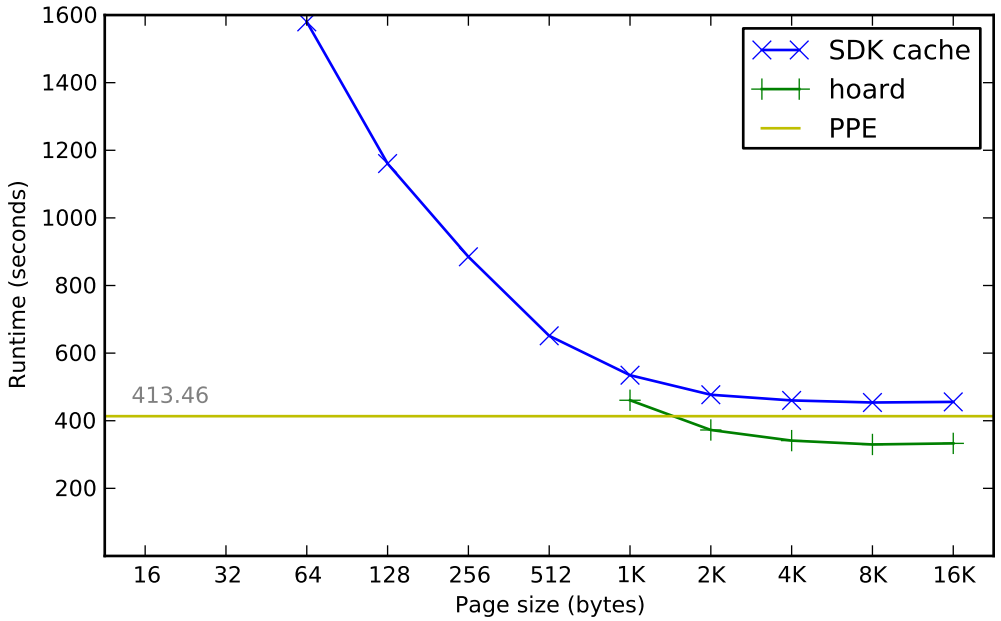
the SDK cache version in all cases.

That the SPE/hoard implementation outperforms the PPE version is surprising as the computation in `179.art` consists almost entirely double-precision floating point arithmetic, a workload that the Cell BE SPE has large penalties in executing (see 4.2.2). The program does, however, have straightforward data access patterns, which helps to explain the benefit of larger pages.

Figure 8.19 illustrates that there is a large difference between the number of DMA gets and puts that the SDK cache performs — as a result, the hoard implementation will be performing many more writes than necessary. It is evident that the number of fetches and writes decreases as the page size increases, which is to be expected from the linear array accesses that the program performs.

Figure 8.20 shows the hit rate improving as the page size increases.

Figure 8.21 shows that the amount of DMA traffic is relatively stable for all of the configurations, increasing by less than 10% for the largest hoard block size. The cause of the disjointed decrease for the smaller SDK cache page sizes is less clear, possibly due to a decrease in conflict misses.



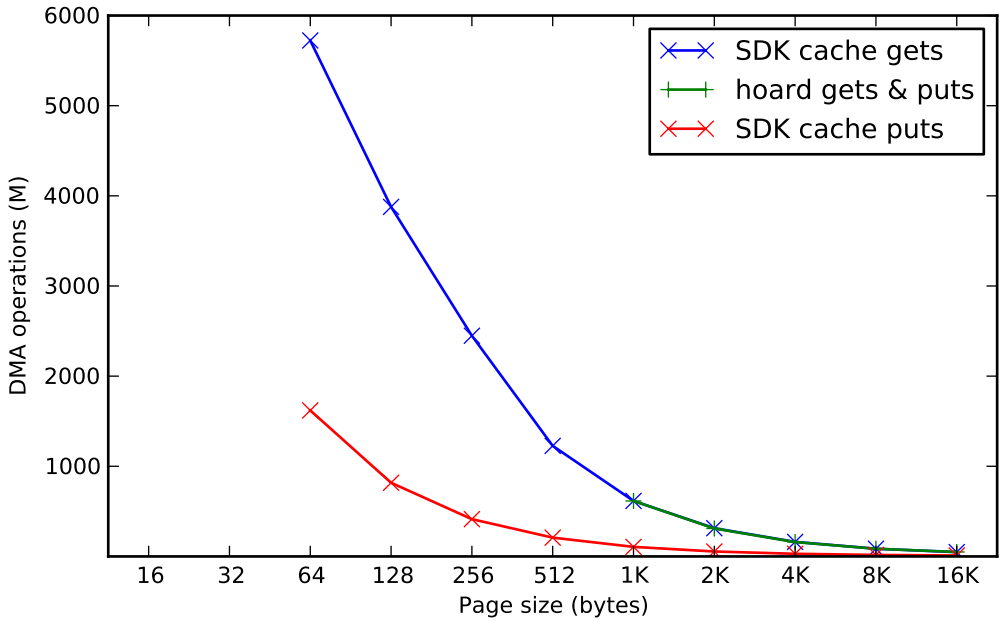
Page size	64	128	256	512	1K	2K	4K	8K	16K
SDK cache	1,580	1,161	886.0	651.2	534.7	476.9	460.1	453.9	455.7
hoard					460.6	372.6	341.2	329.8	333.0

Figure 8.18: Runtimes of hoard and SDK cache for `179.art`

This graph also illustrates one consequence of reducing the write traffic as done by the SDK cache — there is a wide margin between the amount of data transferred by the two memory managers, although it is clear that this does not provide an overall benefit for the SDK cache in terms of runtime.

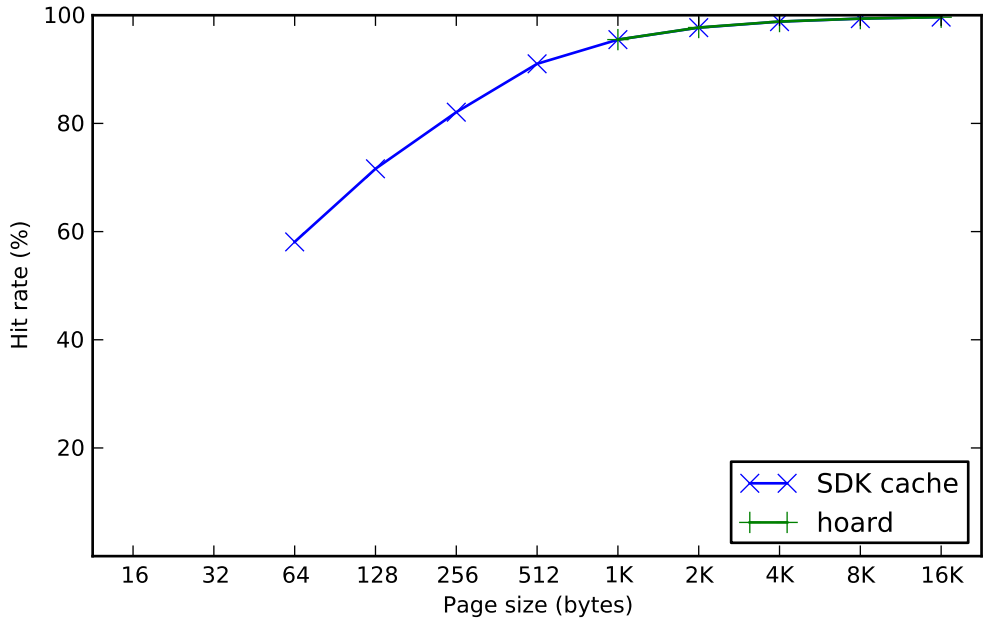
Even with the extra overhead of writing data back to memory, the data throughput when using larger hoard page sizes is not particularly high when compared to the limits of the platform. The higher hit rate of the hoard combines with the shorter runtime to increase the total DMA throughput.

There is a slight above-trend increase in throughput and runtime for the hoard with 16KB pages, indicating that it may be experiencing an increased rate of capacity misses over the smaller-paged alternatives.



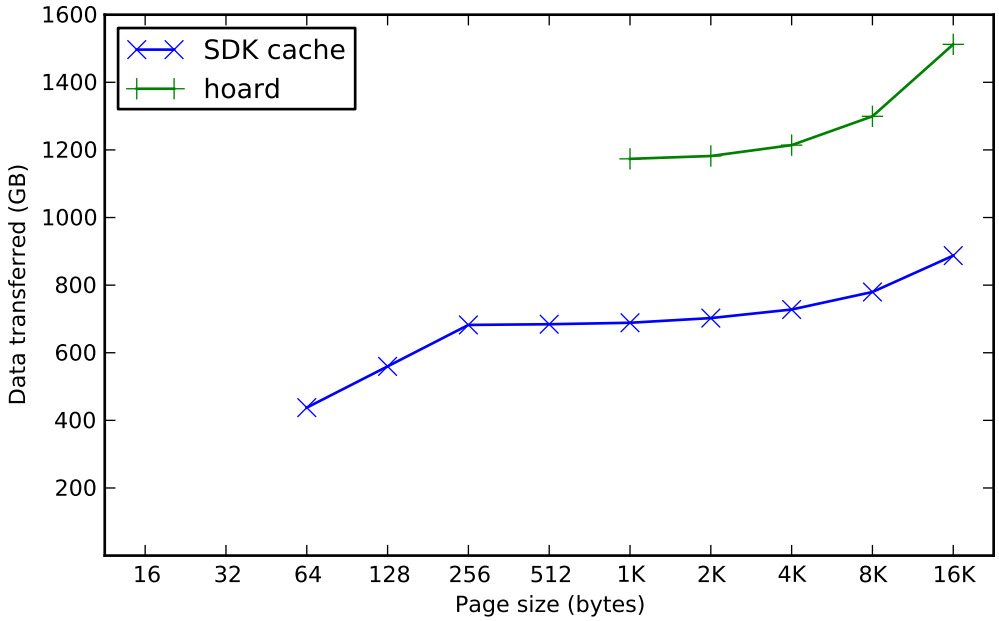
Page size	64	128	256	512	1K	2K	4K	8K	16K
SDK cache gets	5,723	3,877	2,448	1,227	616.0	313.4	161.5	85.55	47.71
hoard gets & puts					615.3	309.9	159.1	85.15	49.55
SDK cache puts	1,620	817.3	413.7	208.6	106.1	54.92	29.36	16.65	10.44

Figure 8.19: Split DMA *get* and *put* counts of hoard and SDK cache for `179.art`



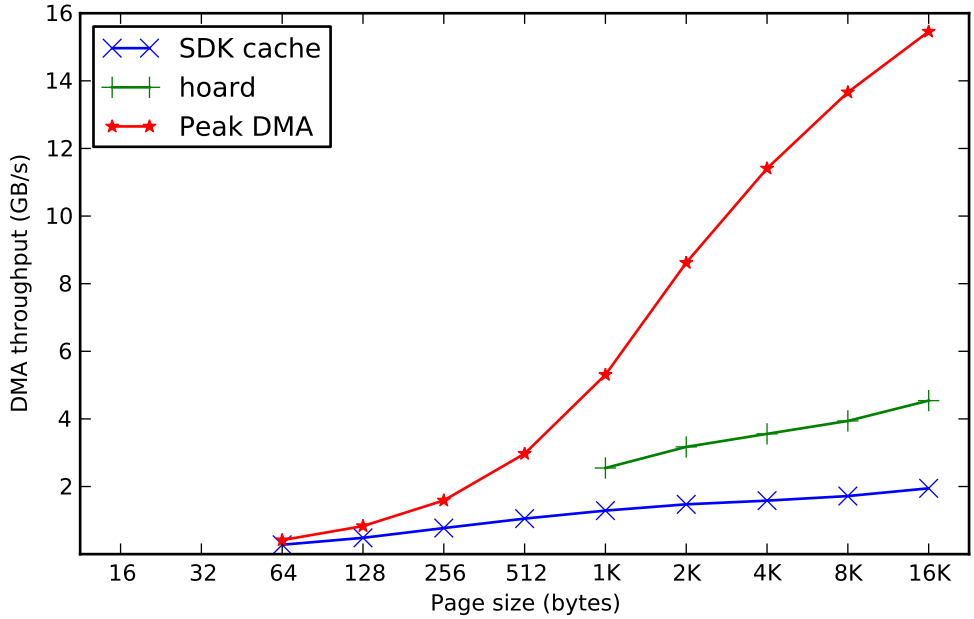
Page size	64	128	256	512	1K	2K	4K	8K	16K
SDK cache	58.09	71.61	82.08	91.02	95.49	97.71	98.82	99.37	99.65
hoard					95.50	97.73	98.83	99.38	99.64

Figure 8.20: Hit rates of hoard and SDK cache for `179.art`

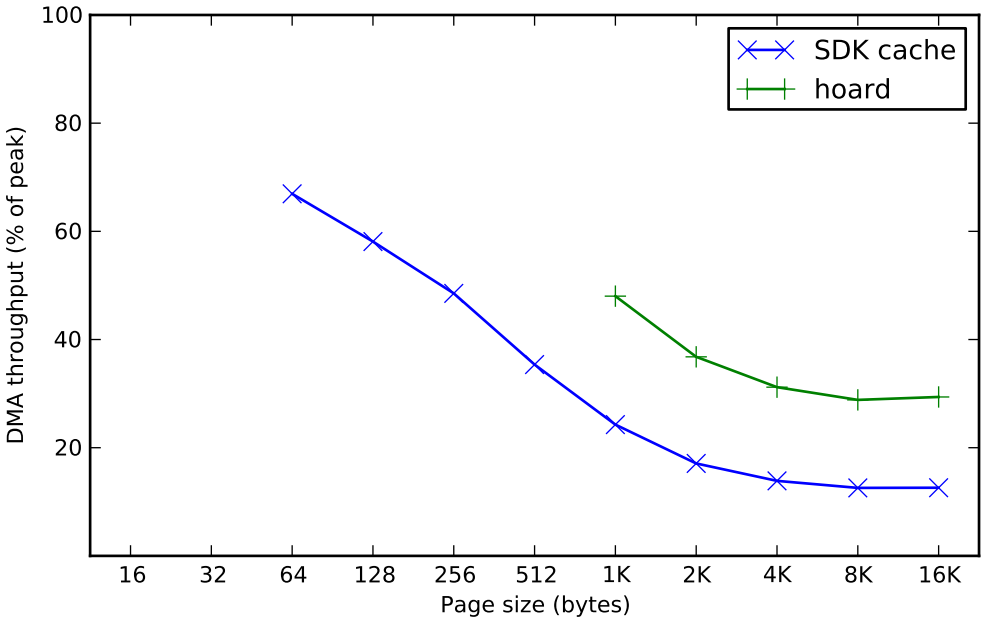


Page size	64	128	256	512	1K	2K	4K	8K	16K
SDK cache	437.7	559.7	682.3	684.3	688.7	702.5	727.9	780	887
hoard					1,174	1,182	1,214	1,299	1,512

Figure 8.21: Total data transfer of hoard and SDK cache for `179.art`



(a) Total average DMA throughput



(b) Average DMA throughput as a percentage of peak

Page size	64	128	256	512	1K	2K	4K	8K	16K
Peak	0.41	0.83	1.59	2.97	5.31	8.62	11.41	13.66	15.46
SDK cache	0.28	0.48	0.77	1.05	1.29	1.47	1.58	1.72	1.95
hoard					2.55	3.17	3.56	3.94	4.54
SDK cache %	66.94	58.11	48.53	35.37	24.27	17.09	13.87	12.57	12.6
hoard %					48.02	36.8	31.19	28.84	29.38

Figure 8.22: Average DMA throughput of hoard and SDK cache for `179.art`

8.4.1 Summary for `179.art`

There are several potential hoard optimisations that may improve performance for this program. The linear array access patterns suggest that pre-fetching memory pages may help reduce the time spent waiting for DMA operations to complete. The difference between the number of pages fetched and the number written by the SDK cache suggests that there may be benefit in keeping track of whether a page has been modified or not and writing back only those that have.

8.5 `181.mcf`

This program is a particular challenge for the memory managers and performs poorly on many different hardware caches (Hennessy & Patterson (2007) summarises a range of different tests of this benchmark). The program involves graph analysis, accessing data through many levels of pointer indirection and thus putting a great deal of pressure on any cache mechanism with its large, unpredictable working set.

The largest structure in `181.mcf` has a size of 64 bytes, which prevents running the program using the SDK cache with a page size of 16 or 32 bytes. The statistics for 16KB page sizes are omitted from the results presented due to the large increase over the other measured values — including them on the graphs obscures the other results. The trend for the omitted results is consistent with the other results present.

Figure 8.23 shows that no hoard configuration is close in performance to the SDK cache, and Figure 8.24 shows a peak hit rate of 88% — approximately one miss for every twelve cache accesses. This is the worst hit rate of the programs tested, and is one of the primary reasons for the poor overall performance of this program when running on the SPE: for every miss, there is no way (in the current implementation) to hide the time delay of fetching the data.

This benchmark also exhibits lower hit rates for the hoard than for the SDK cache, something that may seem unintuitive as the hoard's fully associative lookup can reasonably be expected to yield better hit rates overall. This is caused by the hoard having a smaller number of usable pages available, and has two primary causes:

- *A larger page-table space overhead of the hoard* — The previous benchmarks have smaller program text than `181.mcf`, and as a result there is less local store available for

caching data. While there is sufficient local store space available that the SDK cache programs may have 128KB of usable space, the hoard requires 16KB for its first level page table, and can typically allocate less space due to alignment restrictions. In this case there is 120KB available when using 1KB pages and 104KB with 16KB pages. The SDK cache has 128KB of usable storage for all page sizes.

- *More d-pages when accesses have low locality* — `181.mcf` has a low hit rate and low spacial locality which increases the number (and thus amount of space) needed for d-pages. This results in a further reduction in pages available for data storage.

As a result of these causes, the hit rate of the hoard programs is lower than is the case for the SDK cache.

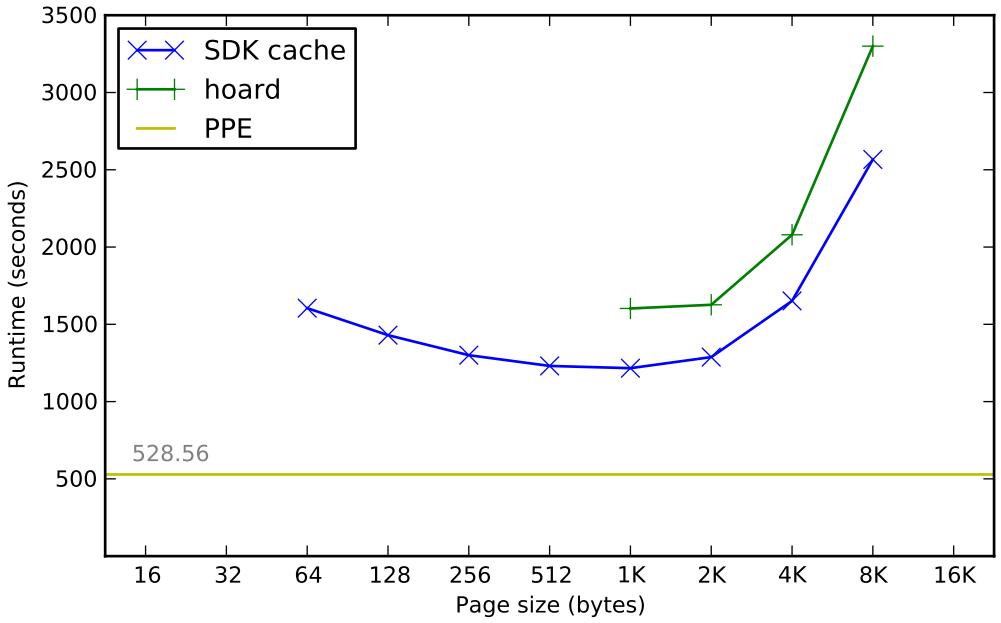
The lower hit rate when using the hoard is apparent in the increased number of fetches that are issued by the hoard over the number issued by the SDK cache, as can be seen in Figure 8.25. It is also clear that there is a very wide gap between the number of pages read and the number that need to be written back to main memory. The DMA overhead of the hoard is almost doubled due to this.

Figure 8.26 shows the large amount of extra data that is transferred for hoard configurations as a result of the lower hit rate and need to write back all pages, and it can be seen in the resultant throughput depicted in Figure 8.27 that most of the runtime of the program is a consequence of the amount of data copied to and from main memory.

8.5.1 Summary for `181.mcf`

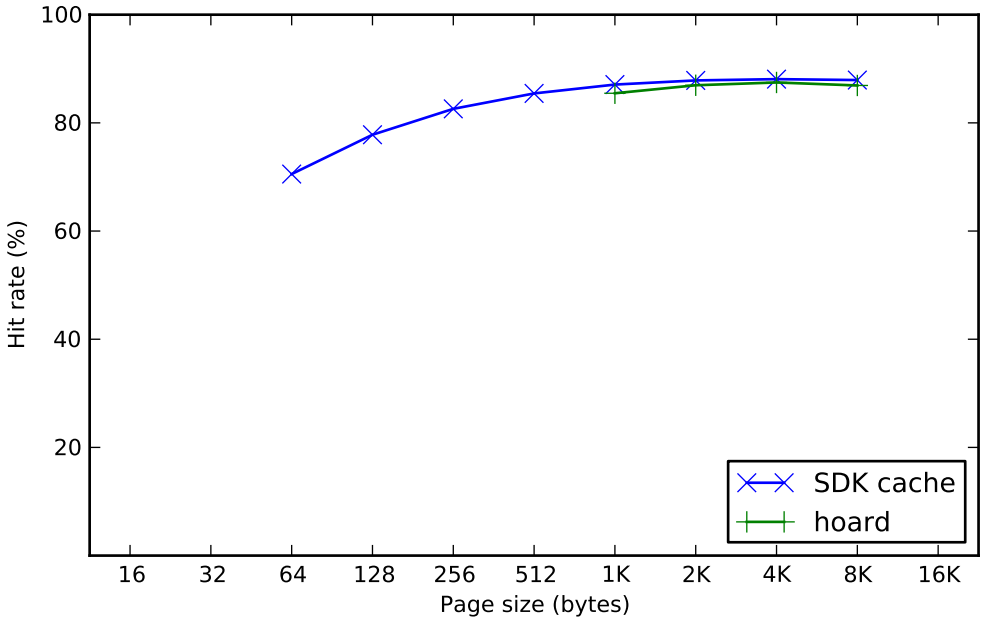
The hoard, as implemented, is poorly suited to running `181.mcf`. There are a number of ways in which the runtime may be improved. As most of the time is spent waiting for DMA completion, keeping one or more pages free to allow reads to be issued sooner is likely to have a strong positive benefit to the program runtime, as would writing back only those pages modified — or even attempting a write-through policy.

The hit rate of the hoard is particularly poor, so improving this is likely to also be of benefit. Resizing the page tables will help increase the number of usable pages for caching. This should help to increase the hit rate. In addition, it may be possible to improve the hit rate with a different page replacement algorithm.



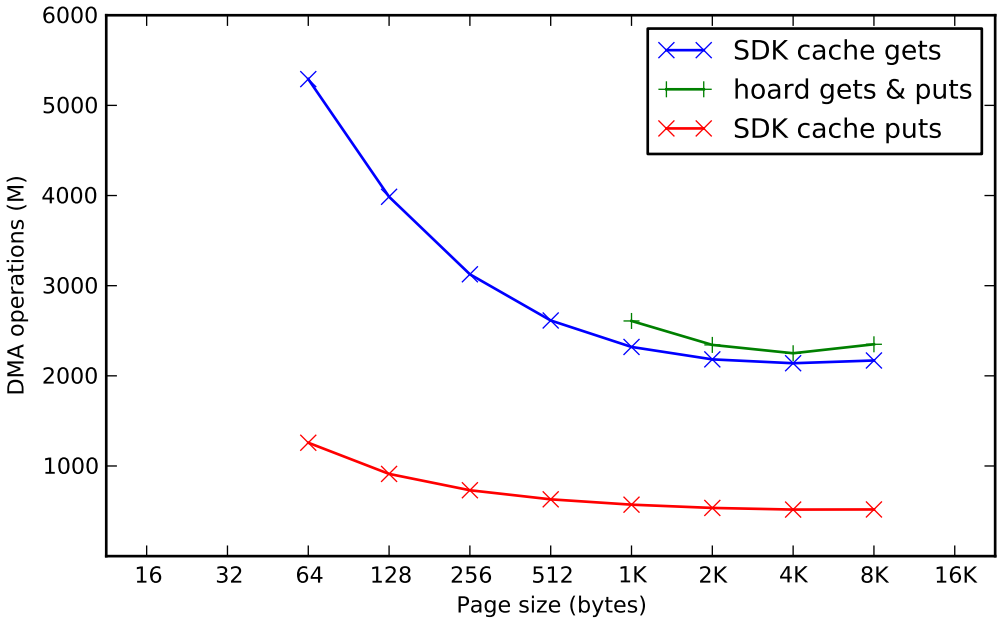
Page size	64	128	256	512	1K	2K	4K	8K
SDK cache	1,605	1,430	1,300	1,231	1,216	1,288	1,652	2,567
hoard					1,603	1,626	2,079	3,300

Figure 8.23: Runtimes of hoard and SDK cache for `181.mcf`



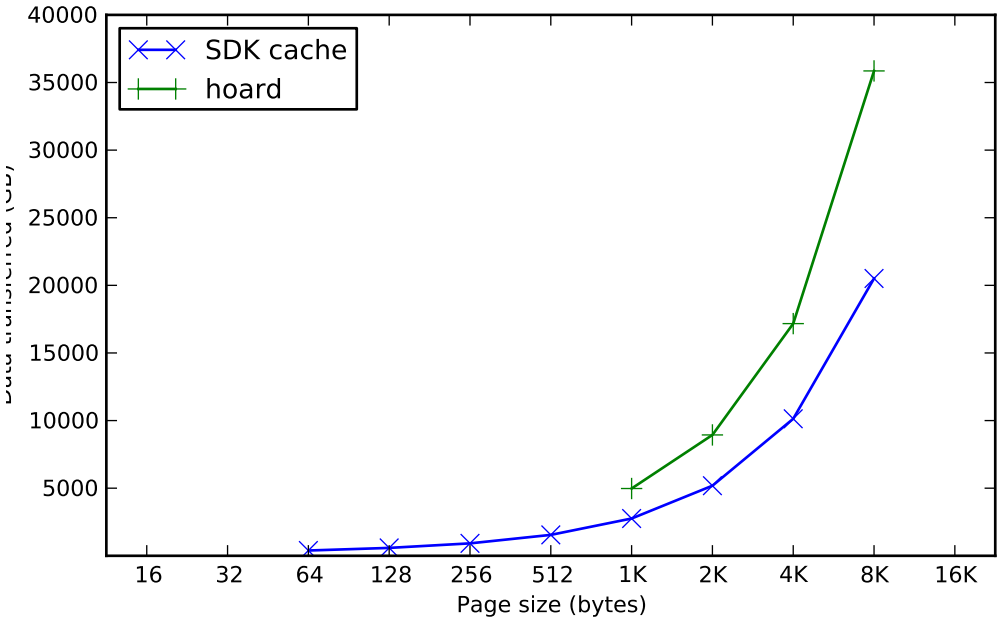
Page size	64	128	256	512	1K	2K	4K	8K
SDK cache	70.54	77.80	82.59	85.44	87.08	87.85	88.09	87.92
hoard					85.47	86.95	87.47	86.92

Figure 8.24: Hit rates of hoard and SDK cache for `181.mcf`



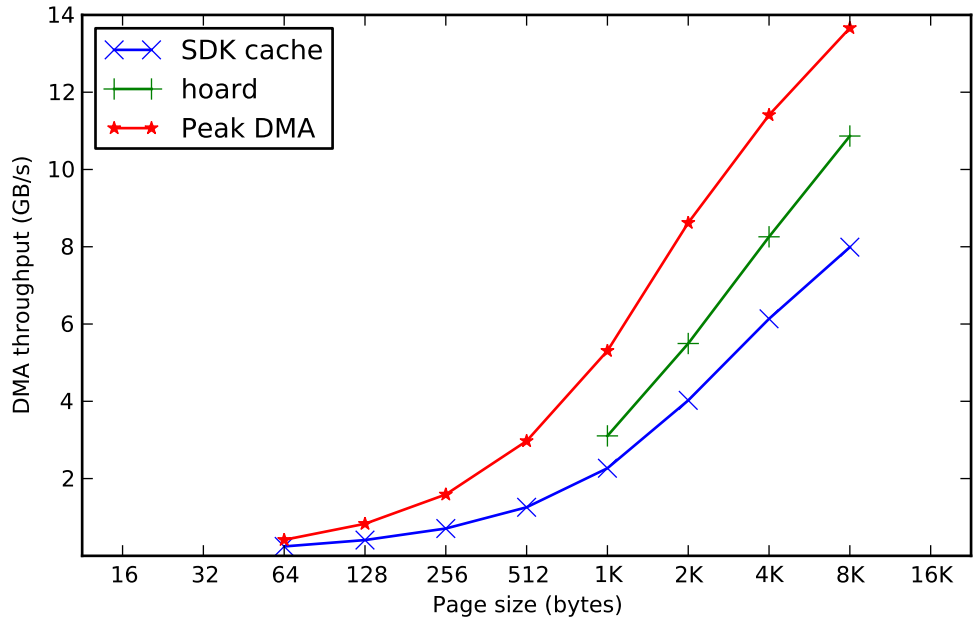
Page size	64	128	256	512	1K	2K	4K	8K
SDK cache gets	5,291	3,986	3,126	2,614	2,320	2,183	2,140	2,170
hoard gets & puts					2,609	2,343	2,251	2,350
SDK cache puts	1,257	912.1	730.9	630.1	570.9	534.6	516.5	518.8

Figure 8.25: Split DMA *get* and *put* counts of hoard and SDK cache for `181.mcf`

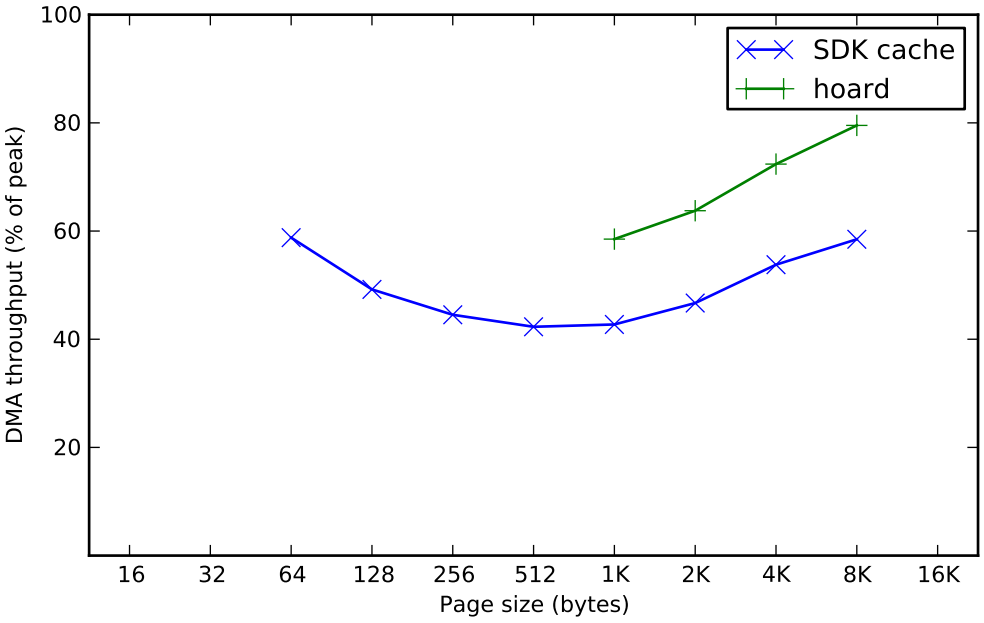


Page size	64	128	256	512	1K	2K	4K	8K
SDK cache	390.4	583.9	919.6	1,547	2,757	5,183	10,133	20,508
hoard					4,976	8,938	17,171	35,857

Figure 8.26: Total data transfer of hoard and SDK cache for `181.mcf`



(a) Total average DMA throughput



(b) Average DMA throughput as a percentage of peak

Page size	64	128	256	512	1K	2K	4K	8K
Peak	0.41	0.83	1.59	2.97	5.31	8.62	11.41	13.66
SDK cache	0.24	0.41	0.71	1.26	2.27	4.02	6.14	7.99
hoard					3.10	5.50	8.26	10.87
SDK cache %	58.78	49.21	44.52	42.3	42.73	46.69	53.78	58.48
hoard %					58.51	63.76	72.38	79.53

Figure 8.27: Average DMA throughput of hoard and SDK cache for `181.mcf`

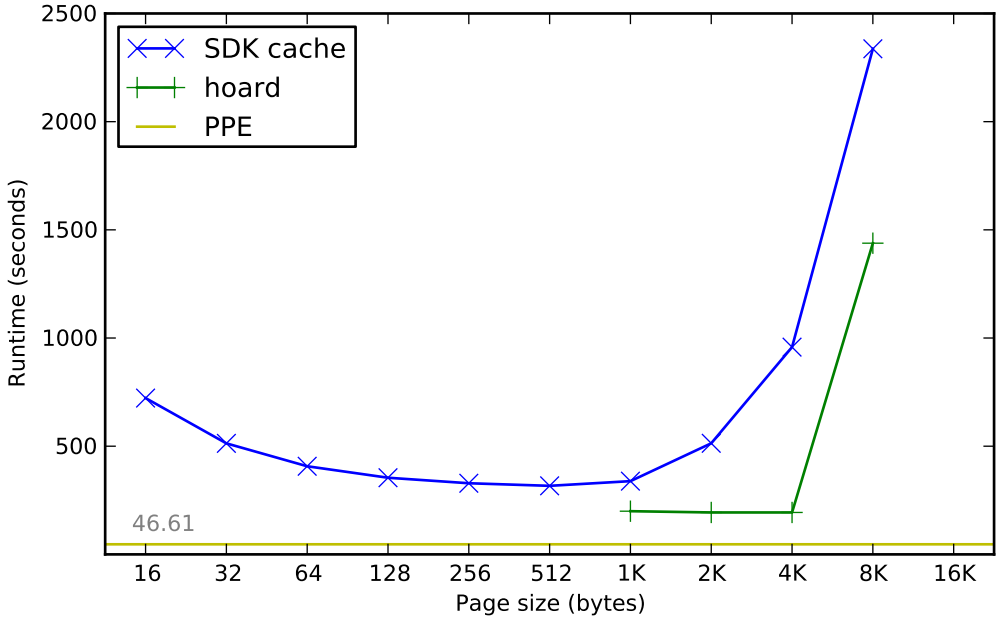
8.6 183.equake

Like 179.art, 183.equake performs a large amount of double precision arithmetic in the form of large matrix multiplications. The program size is somewhat larger than for other benchmarks used so far and the SDK cache size has been reduced from 128KB to 64KB to fit the available local store. The size of hoard pages ranges from 108KB for 1KB pages to 96KB for 8KB pages.

Results for the 16KB page size configurations have been omitted as they follow the trend of the preceding points and including them makes the graphs much less readable.

For these tests, the SPEC2000 train dataset has been used as it is much smaller and faster to complete than the ref data-set, but yields a similar pattern of results.

An interesting pattern of runtimes for this program can be seen in Figure 8.28, particularly for the hoard where there is a very sharp 'elbow' in the runtime line. For the points to the left of that, a trend downwards may be observed, indicating benefits of the increased DMA efficiency when using larger pages, while the sharp jump would seem to indicate that there are not enough pages to hold the program's working set. Performance suffers



Page size	16	32	64	128	256	512	1K	2K	4K	8K
SDK cache	722.5	513.1	407.3	354.3	328.8	316.9	338.5	513.5	958.0	2,337
hoard							199.6	193.8	193.8	1,438

Figure 8.28: Runtimes of hoard and SDK cache for 183.equake

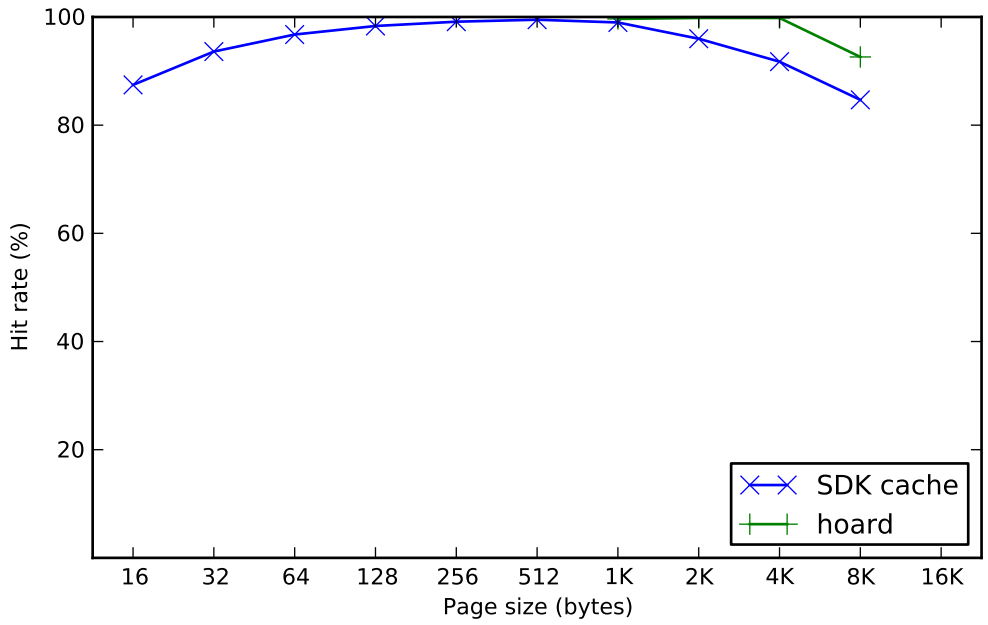
greatly as a result.

The overall runtime is slower when using the SDK cache, and the change in runtime is more gradual. This may attributed to the smaller cache size (the 64KB of the SDK cache is smaller than any of the hoard configurations) as well as its set associative design, which will experience a higher rate of conflict misses.

Both SPE programs are much slower than the PPE, by a larger factor than has been the case for previous programs. The fastest measured runtime on the SPE is more than four times longer.

Figure 8.29 clearly shows a very high hit rate for hoard page sizes from 1 to 4KB (peaking at around one miss in 500 access), and the hit rate for the SDK cache with a page size of 512 bytes is also very high (approximately one miss in 200 accesses).

Figure 8.30 again shows quite an interesting set of results for the hoard. The total number of reads for the hoard is less than that for the same sized SDK setting, and the SDK cache indicates that only a small percentage of pages loaded are modified before they are written back — but the hoard performs less reads and writes combined than the SDK cache performs writes for 4KB pages. The working set of pages clearly fits in the hoard and is



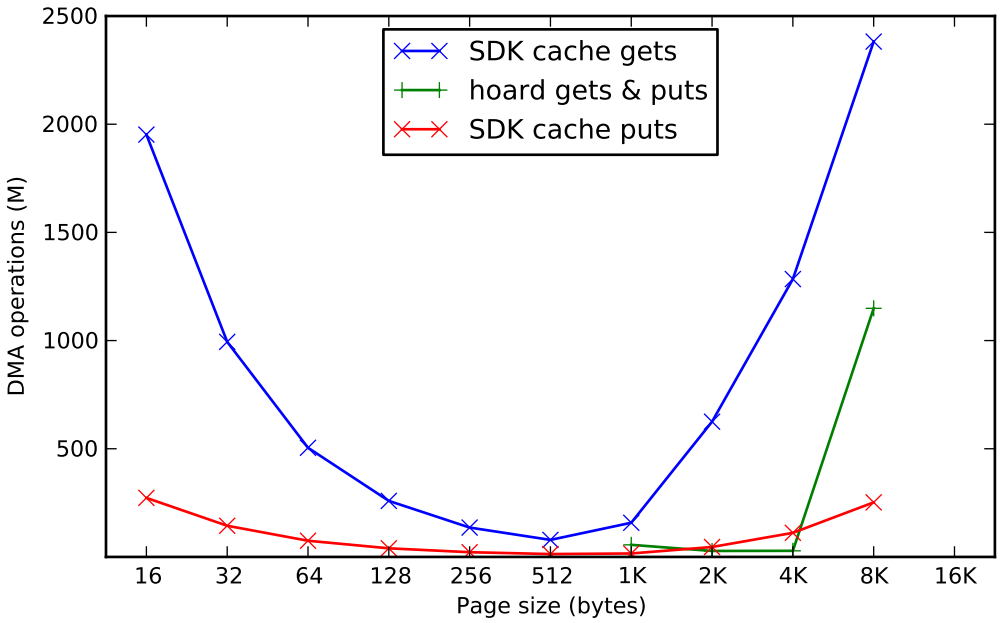
Page size	16	32	64	128	256	512	1K	2K	4K	8K
SDK cache	87.44	93.61	96.75	98.33	99.13	99.49	98.98	95.98	91.74	84.68
hoard							99.64	99.82	99.82	92.61

Figure 8.29: Hit rates of hoard and SDK cache for `183.equake`

being used very efficiently with this configuration.

In Figure 8.31 it can be seen that the total amount of data transferred stays quite low when the working set fits in local store. The increase for software cache begins for smaller pages due to the lower effectiveness of the set associative form, but the increase for 8KB hoard pages is very sharp.

In terms of minimising the amount of data transferred, the smallest pages are clearly the most efficient, but it remains clear that transferring less data is not the way to achieve maximum program performance on this architecture due to the high DMA bandwidth available.

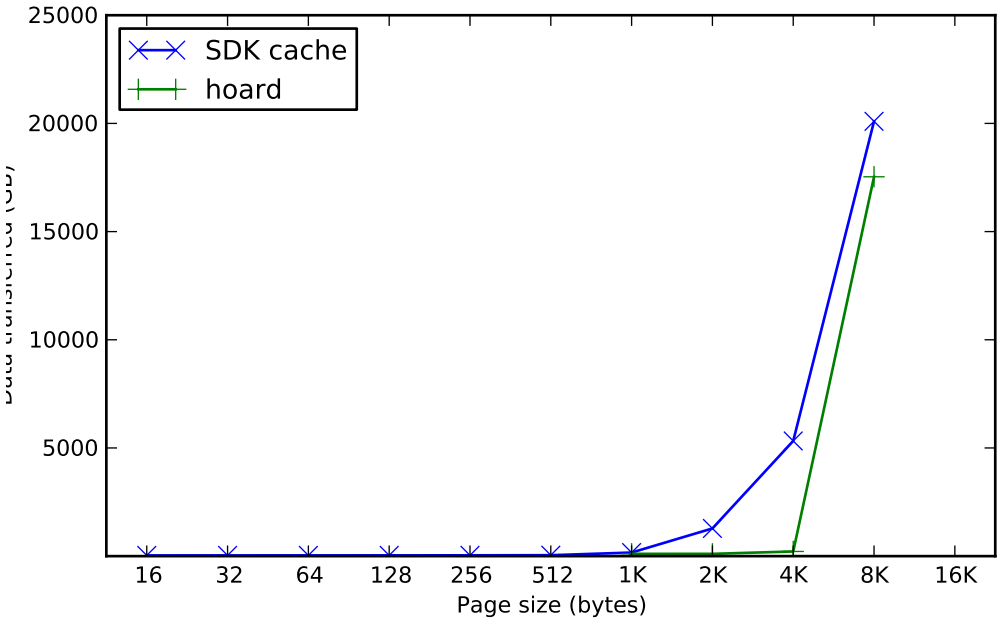


Page size	16	32	64	128	256	512	1K	2K	4K	8K
SDK cache gets	1,952	994.0	504.6	259.1	135.3	78.8	158.0	625.2	1,284	2,382
hoard gets & puts							55.93	27.67	28.27	1,149
SDK cache puts	272.9	144.1	74.83	39.83	21.82	13.34	15.93	45.92	111.4	252.1

Figure 8.30: Split DMA *get* and *put* counts of hoard and SDK cache for `183.equake`

As for all the graphs of `183.equake` the throughput shown in Figure 8.32 makes clear the stark contrast between the case when the working set fits in local store and when it doesn't. The throughput in the more optimal cases is only a small fraction of the peak DMA for that page size.

The rate of throughput for 8KB pages is worth noting — the hoard program with this configuration runs faster, but the SDK cache exhibits lower throughput. Higher throughput and lower runtime can be seen, to a lesser extent, in most of the preceding benchmarks.



Page size	16	32	64	128	256	512	1K	2K	4K	8K
SDK cache	33.16	33.92	34.53	35.63	37.58	43.92	165.9	1,280	5,324	20,096
hoard							106.7	105.6	215.7	17,533

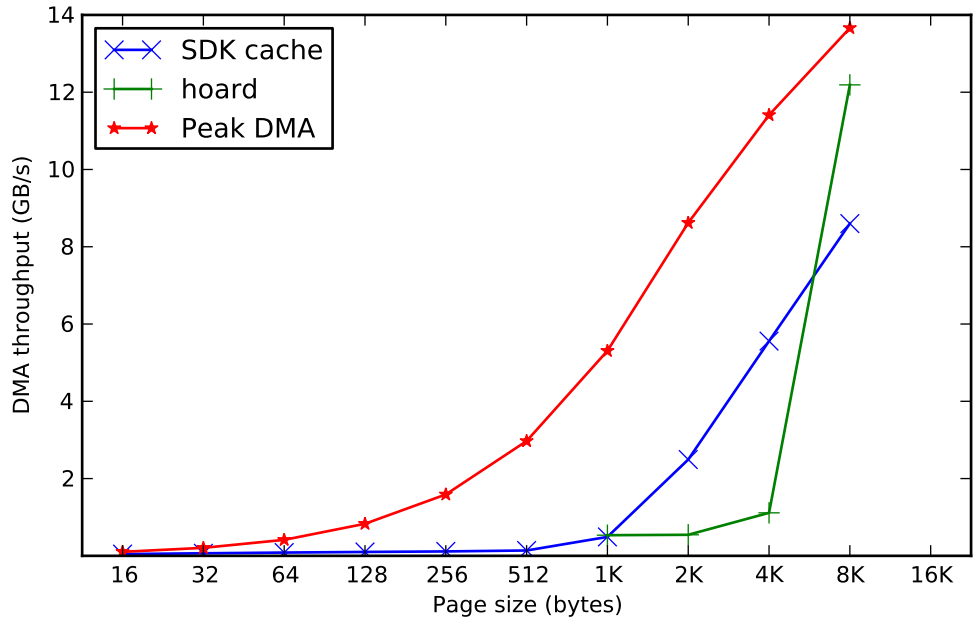
Figure 8.31: Total data transfer of hoard and SDK cache for `183.equake`

8.6.1 Summary for `183.equake`

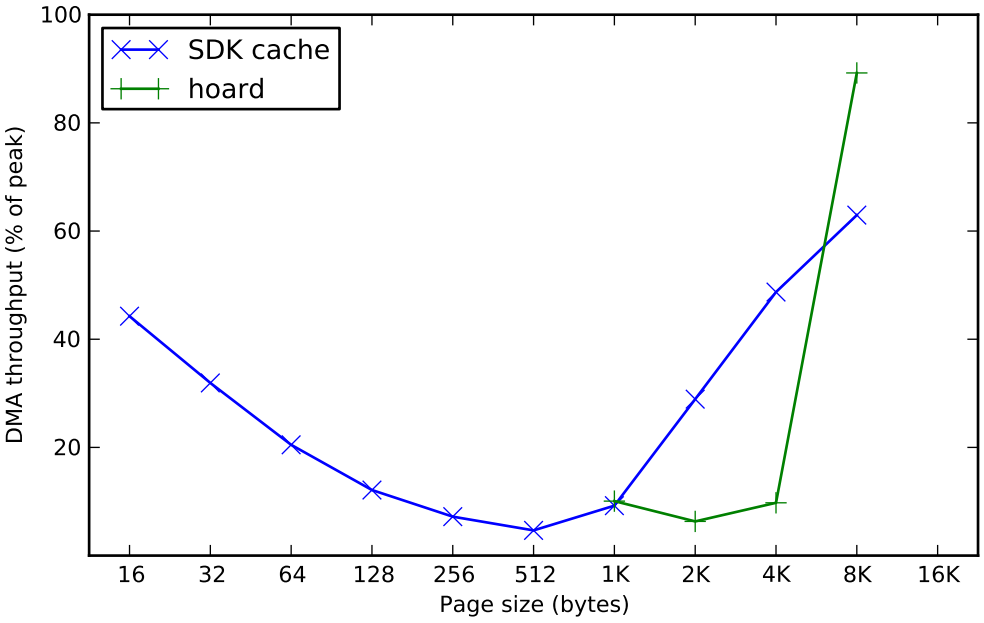
The opportunities for optimising hoard performance on this benchmark seem limited — for the fastest configurations, only a small percentage of time is spent waiting for DMA completion, so there is not a lot to be gained by reducing this delay. Options like pre-fetching, maintaining free page(s) and writing only dirty pages do not seem likely to yield large benefits.

A different page replacement algorithms may improve the performance, particularly for larger pages, if it is possible to keep a small working set in local store.

The very high hit rate suggests that the speed of access to cached data may be one of the biggest bottlenecks for this program.



(a) Total average DMA throughput



(b) Average DMA throughput as a percentage of peak

Page size	16	32	64	128	256	512	1K	2K	4K	8K
Peak	0.10	0.21	0.41	0.83	1.59	2.97	5.31	8.62	11.41	13.66
SDK cache	0.05	0.07	0.08	0.10	0.11	0.14	0.49	2.49	5.56	8.60
hoard							0.53	0.54	1.11	12.19
SDK cache %	44.26	31.92	20.48	12.12	7.2	4.66	9.23	28.92	48.72	62.95
hoard %							10.07	6.32	9.76	89.23

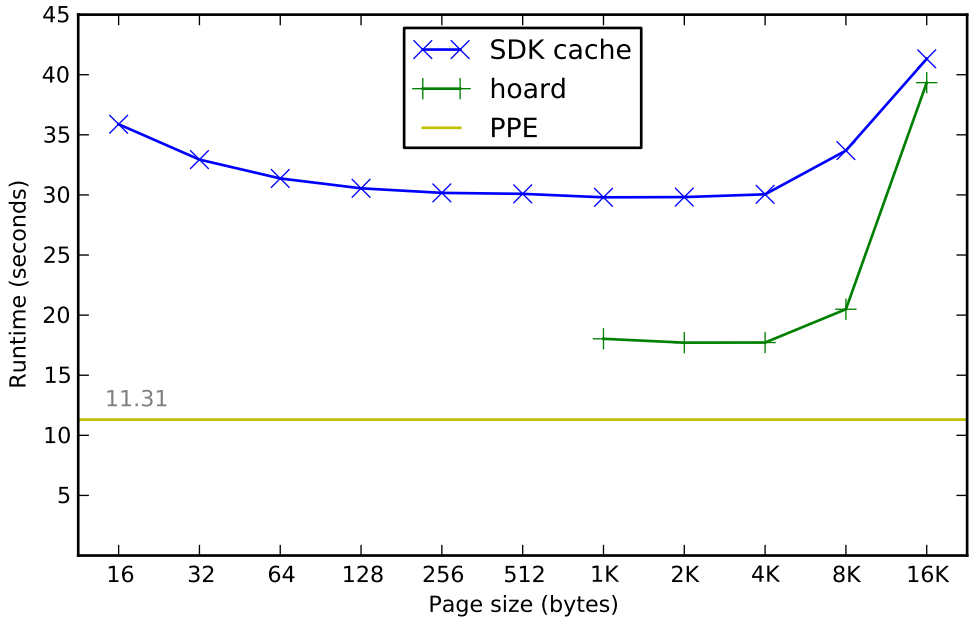
Figure 8.32: Average DMA throughput of hoard and SDK cache for `183.equake`

8.7 mpeg2decode

The benchmark used for this program is the decoding of a 57 second 320 × 240 pixel MPEG1 video stream. It consists of a 2000 kbps video and a 128 kbps audio streams, combined as 266 kilobytes/sec needing to be processed by the decoder.

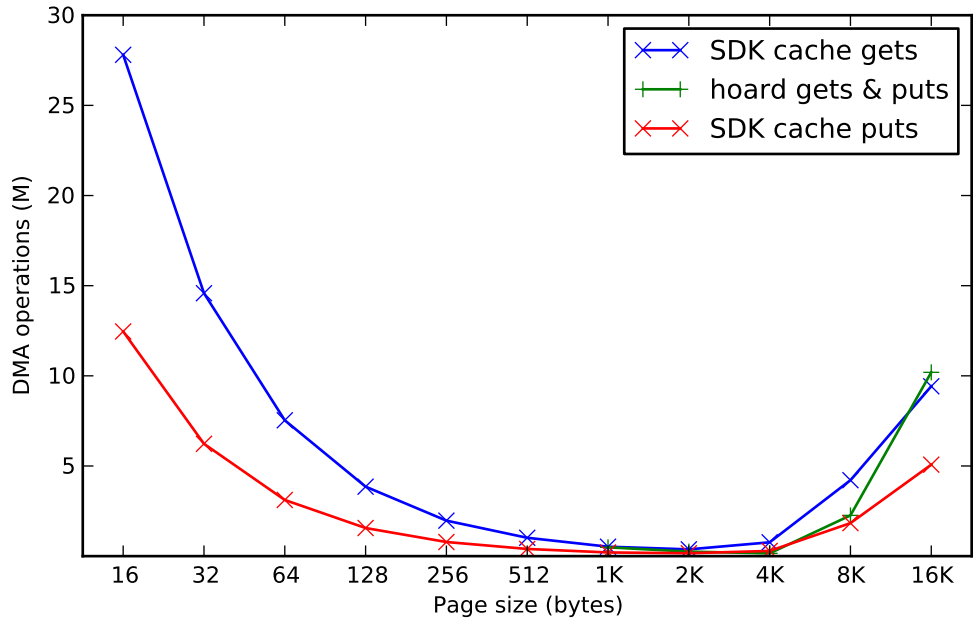
The decoding of each frame depends on the data from previous frames, the number of previous frames varying between one and twenty, depending on the amount of action on screen. In general, it can be expected that the data for a single frame of this video will fit in local store.

Figure 8.33 shows that all configurations are able to decode this video fast enough for it to be displayed in real time. The program runs fastest on the PPE and slowest with the SDK cache with the fastest SPE configuration using the hoard with 2KB page size, running approximately 57% slower than the PPE version. There seems to be a working set effect affecting the results, as performance increase with page size until a page size of 8KB is reached.



Page size	16	32	64	128	256	512	1K	2K	4K	8K	16K
SDK cache	35.88	32.94	31.37	30.55	30.17	30.09	29.80	29.82	30.05	33.69	41.32
hoard							18.03	17.71	17.72	20.49	39.34

Figure 8.33: Runtimes of hoard and SDK cache for `mpeg2decode`



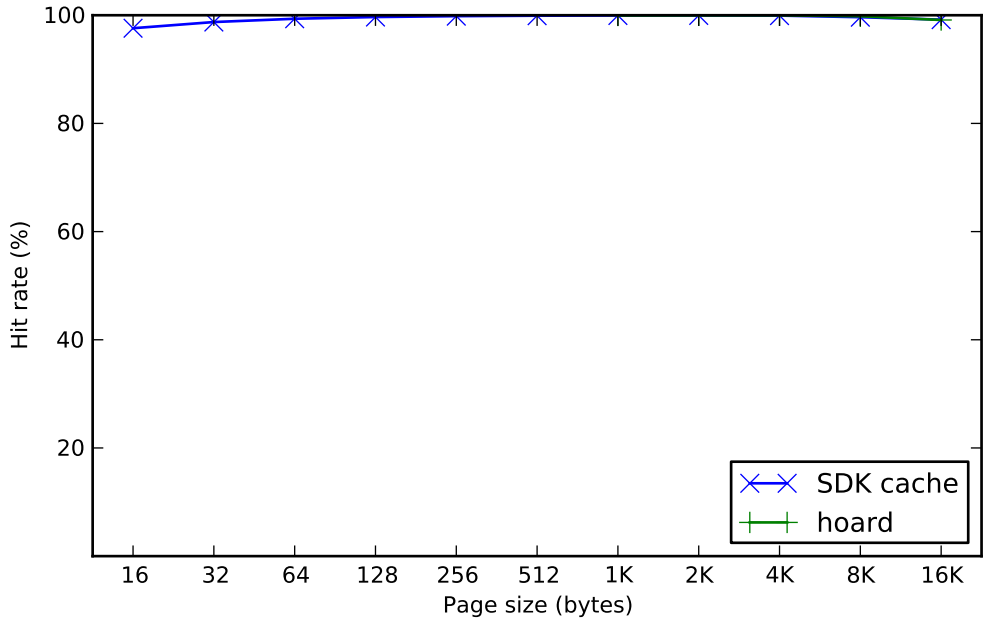
Page size	16	32	64	128	256	512	1K	2K	4K	8K	16K
SDK cache gets	27.80	14.58	7.53	3.85	1.97	1.02	0.52	0.38	0.77	4.21	9.42
hoard gets & puts							0.49	0.25	0.15	2.26	10.19
SDK cache puts	12.46	6.23	3.12	1.56	0.78	0.40	0.20	0.16	0.29	1.83	5.07

Figure 8.34: Split DMA *get* and *put* counts of hoard and SDK cache for `mpeg2decode`

Hit rates stay high for all SPE configurations (as per Figure 8.35), but as has been seen in other benchmarks, even seemingly small decreases in hit rate with large pages results in large increases in DMA traffic and decreases in performance.

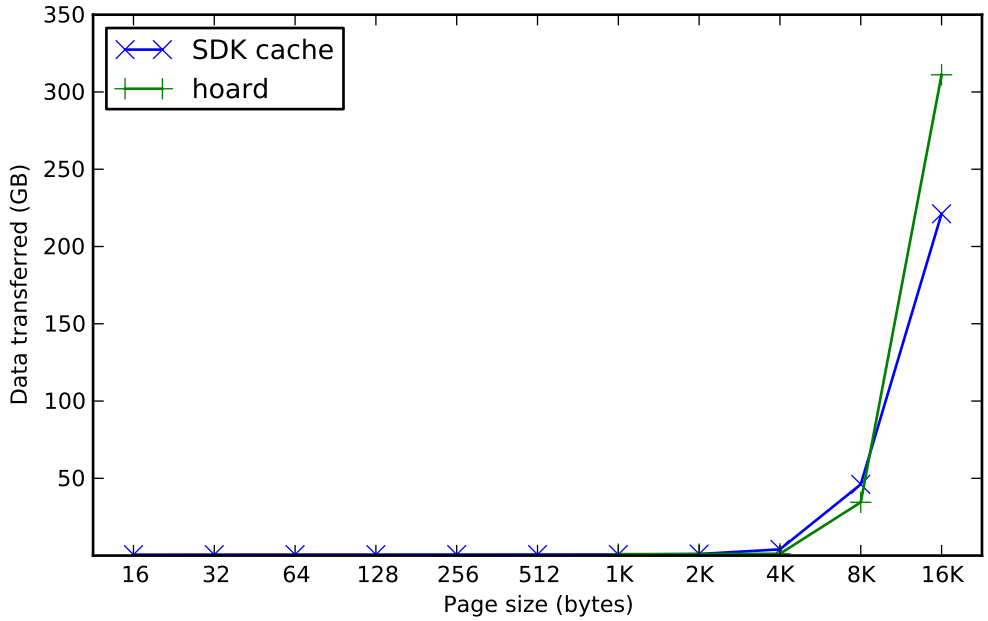
There is a gap between the number of DMA gets and puts for this program. The amount of data transferred for all SDK cache configurations is in excess of 600MB when the actual video file is only 16MB. Regardless, 600MB fetched in a runtime that exceeds thirty seconds is only a small fraction of what the architecture is capable of.

There is little variation in the amount written, up to a 1KB page size, and it is very close to the decoded size of the video, which is approximately 200MB. The hoard is still writing out every page, but this does not add a much to the runtime while the working set fits in local store. The loading of pages to write out affect the amount of data that needs to be read — each page written indicates another page that has been replaced at some stage. Changing the program to have a different write policy (similar to `julia_set`) would likely reduce the contention in the cache, and result in fewer capacity misses.



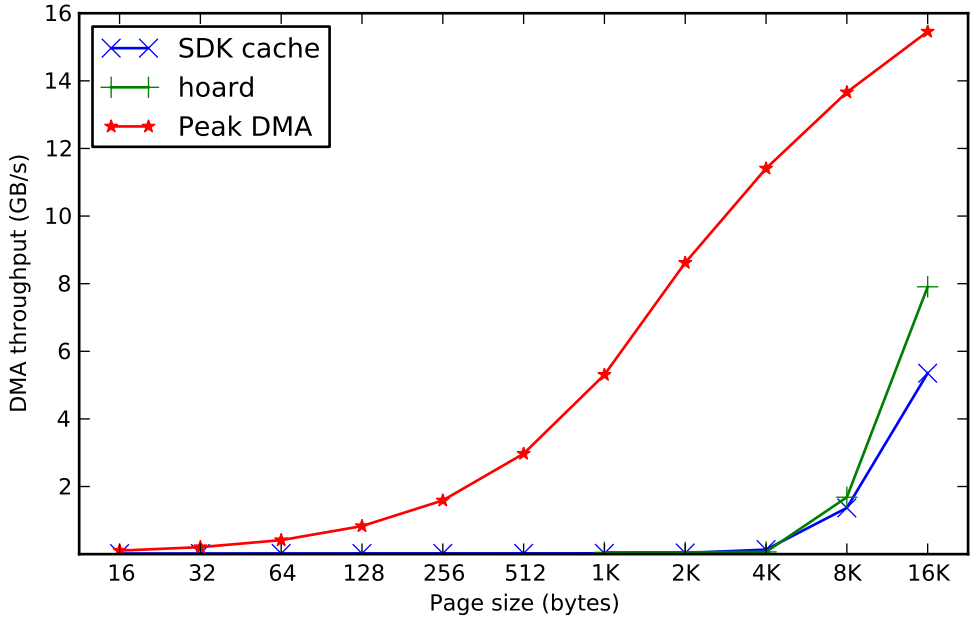
Page size	16	32	64	128	256	512	1K	2K	4K	8K	16K
SDK cache	97.60	98.74	99.35	99.67	99.83	99.91	99.95	99.97	99.93	99.64	99.19
hoard							99.96	99.98	99.99	99.80	99.12

Figure 8.35: Hit rates of hoard and SDK cache for `mpeg2decode`

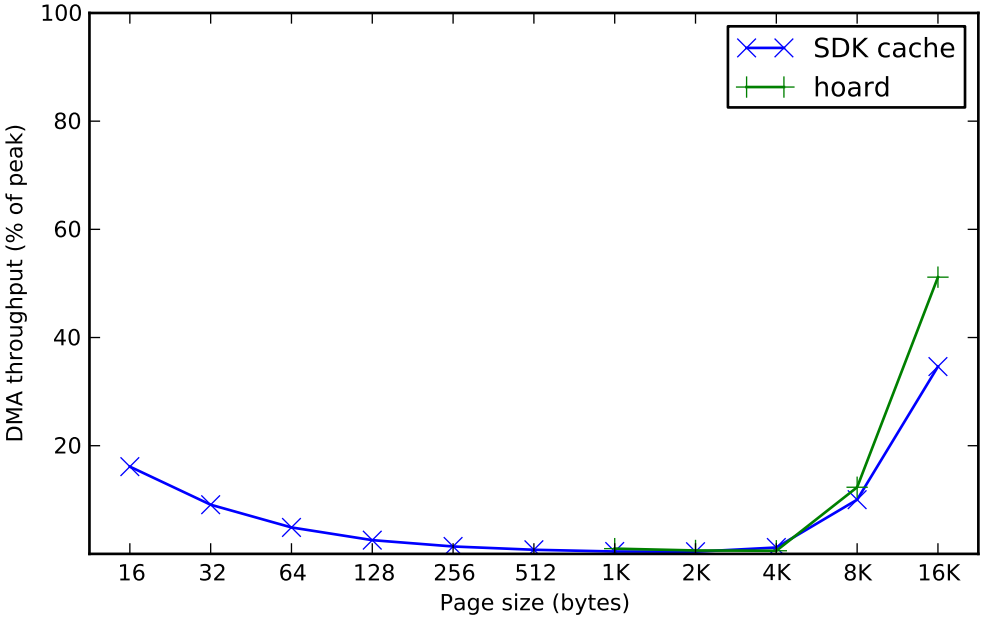


Page size	16	32	64	128	256	512	1K	2K	4K	8K	16K
SDK cache	0.60	0.62	0.63	0.65	0.66	0.68	0.69	1.02	4.03	46.11	221.16
hoard							0.93	0.96	1.18	34.49	311.10

Figure 8.36: Total data transfer of hoard and SDK cache for `mpeg2decode`



(a) Total average DMA throughput



(b) Average DMA throughput as a percentage of peak

Page size	16	32	64	128	256	512	1K	2K	4K	8K	16K
Peak	0.10	0.21	0.41	0.83	1.59	2.97	5.31	8.62	11.41	13.66	15.46
SDK cache	0.02	0.02	0.02	0.02	0.02	0.02	0.02	0.03	0.13	1.37	5.35
hoard							0.05	0.05	0.07	1.68	7.91
SDK cache %	16.13	9.09	4.89	2.54	1.37	0.76	0.44	0.4	1.18	10.02	34.63
hoard %							0.97	0.63	0.58	12.32	51.16

Figure 8.37: Average DMA throughput of hoard and SDK cache for `mpeg2decode`

When the page size reaches 16KB the hoard is not only slower, but results in more page misses, the result of there being only five 16KB pages available.

The total amount of DMA traffic remains very low for all but the largest two page sizes, increasing sharply as the working set no longer fits in local store.

Like `183.equake`, the throughput of the hoard increases above that of the SDK cache, while the hoard remains the faster-running of the two options, although not by much.

8.7.1 Summary for `mpeg2decode`

As the working set for each of these benchmarks fits comfortably in local store, it is clear that this is not a benchmark that is bound by DMA throughput. Rather, it is bound by the rate at which it can perform its data processing, which is clearly much slower than the PPE alternative. While it is true that this benchmark will run fast enough to render the video stream for viewing, it is rare already that such a low-resolution video is of interest on a platform lauded for its high-definition playback abilities. This decoder will clearly not scale up usefully to current high-definition video formats.

If DMA is not a bottleneck, it is worth considering why performance is not higher. One reason may be that cached data is accessed almost exclusively as byte, which does not result in efficient code generated for the SPE. The main reason for poor performance may be the number of calls to `read()` and/or `fread()` that are grossly inefficient when run on the SPE. The overhead of these PPE callbacks is likely to be a big contributor to the difference in runtime between the SPE and PPE implementations.

The difference between the hoard and SDK cache implementations, can be attributed to the faster lookup for the hoard — the SDK cache has a common infrastructure for PPE callbacks and neither performs many DMA operations. The very high hit rate means that any improvement in time to handle a hit will make a large difference to runtime. In this way, `mpeg2decode` is like `183.equake`.

8.8 Interim Conclusions

For higher hit rate programs, the hoard is the faster solution, for lower hit rate programs, the SDK cache yields lower runtimes. The hoard offers faster access times (for hits) and better cache utilisation when a program's working set fits in local store.

From this it can be seen that the relative performance of the hoard can be seen to be strongly correlated with locality of access of a given program — higher locality in memory accesses result in higher hit rates, and the hoard's fully-associative cache provides a clear benefit when the working set can remain in local store as the page size increases, as can be seen particularly for `183.equake`.

But the operation of the hoard includes writing back of all pages and support for page sizes $\geq$ 1KB, and there appears to be a big performance penalty in the form of DMA latency. Additionally, the selection of page-table sizes when using 1KB data pages can excessively decrease the number of pages available for data, resulting in performance decreases.

In the following chapter, the shortcomings of the hoard identified in this chapter will be given further consideration and potential solutions tested and analysed.

8.9 Comparison with COMIC

The source code for COMIC (Lee et al. 2008) became available late in this research, and while there is no direct comparison between the performance of it and the hoard, an analysis of COMIC and consideration of probable performance differences is included here.

COMIC addresses a broader range of problems in Cell BE development than the hoard, particularly providing an alternative to OpenMP style program annotation where it appears to offer significant performance benefits over other alternatives. The design and evaluation of the hoard presented in this research has focussed on single-threaded programs, with a goal of minimal modification of the existing program code. The one parallel program tested performs read-only memory access to hoard-managed data.

As such, a meaningful comparison of COMIC with the hoard as it has been considered in this research must disregard a large amount of the features and value provided by the COMIC system.

The programming models of the two systems are quite different. COMIC requires a PPE-thread to manage SPE threads and their memory usage, where the hoard may be used standalone on a single SPE.

Where the hoard provides a C++ smart pointer class that is able to replace most common uses of pointers in existing code, the COMIC API consists of a range of C functions. In this way, COMIC is more like the SDK cache and is visibly intrusive into written code. It seems

likely that it may be possible to implement a similar `hoard_ptr` type wrapper on top of COMIC's API, as was done for the SDK cache. COMIC uses a four-way set associative cache that is not dissimilar to that of the SDK cache.

Figure 8.38 is based on `COMIC_spe_read_LSA()` from `comic_spe.h` distributed as part of `comic-0.8.1`. It was compiled with `spu-elf-gcc-4.6.0 -O3`[3]. Comparing this output to Figures 5.15 and 5.17 shows that COMIC's lookup routine suffers the same penalties due to complexity and inter-instruction dependency as the SDK cache, while also not being well optimised[4].

Cycle	*Insn.*	*Operands*
–	ila	$13,66051
–	lrq	$14,COMIC_spe_info+80
–	ila	$7,COMIC_spe_info
–	lqr	$5,COMIC_spe_info+96
–	il	$80,1023
–	ila	$81,cache_mem
0	shufb	$3,$3,$3,$13
1	shufb	$6,$14,$14,$13
2	shufb	$4,$5,$5,$13
5	rotm	$6,$3,$6
6	and	$5,$3,$4
9	andi	$4,$6,31
11	ori	$3,$4,0
13	ai	$12,$3,8
15	shli	$11,$12,4
19	lqx	$10,$11,$7
25	ceq	$9,$6,$10
27	gb	$2,$9
29	clz	$2,$2
31	ceqi	$8,$2,32
33	brnz	$8,.L124
34	ai	$18,$3,72
35	shli	$15,$2,2
36	shli	$17,$18,4
40	lqx	$16,$17,$7
46	a	$7,$16,$5
48	rotqby	$2,$7,$15
52	lqd	$3,0($2)

Figure 8.38: COMIC lookup of qword — generated assembly

As a result, it would seem that when using COMIC in a similar way to the hoard — with no program transformation beyond substitution of memory accesses, and no further

[3]Not the same compiler used for the rest of the research, but differences in generated code should be negligible.
[4]A trivial code reorganisation in that function appears to allow GCC to generate significantly better code

optimisation — and based on the results observed for the hoard, program performance may have the following characteristics:

- where programs exhibit a high degree of locality in memory accesses, the faster lookup of the hoard should result in lower program runtimes due to its faster lookup routine;
- where programs exhibit a lower degree of locality in memory accesses, COMIC's more sophisticated index management and support for smaller page sizes should result in runtimes lower than the hoard or SDK cache, although interaction with the PPE for page access may incur some extra delay.

The design of COMIC suits the provision of more than one mechanism for accessing data outside of an SPE's local store, and already supports dynamic changes to page size. It seems quite possible that a paged virtual memory option with faster high-locality random access could be a useful addition to COMIC.

Chapter 9

Optimisation

In Chapter 8, a number of possible improvements to the hoard were identified based on the performance results and comparison with the SDK cache. These were:

- write only modified pages back to main memory, not every loaded page (`179.art`, `181.mcf`);
- asynchronously pre-fetch nearby pages (`qsort`, `julia_set`, `179.art`);
- asynchronously pre-write pages in advance of their replacement (`hsort`, `181.mcf`);
- use a write-through write policy (`hsort`, `181.mcf`);
- perform 'desired word first' style page fetching (`hsort`);
- try different replacement algorithms (`hsort`, `181.mcf`, `183.equake`);
- tune the hoard configuration to the specific memory requirements of the program (`hsort`, `julia_set`, `181.mcf`); and
- optimise interaction with the system (`mpeg2decode`).

Some of these changes will stand alone, others will interact in unpredictable ways, and some are clearly dependent — e.g. many replacement algorithms depend on information about whether a page has been modified (and even if it has been accessed).

To prepare to implement and test these optimisations, it was useful to sort them into functional groupings, which provided a convenient order for implementing and testing each.

1. **Hoard policy**

   Optimisations relating to the algorithms and the way that the hoard manages cached data, which may be grouped in the following ways:

   (a) *Write policy*

   Changes relating to the decisions made when writing out data pages, including writing modified pages, pre-writing pages, and write-through of changes.

   (b) *Fetch policy*

   Changes relating to when and how data is fetched from main memory to local store. Includes pre- and partial-fetching of pages.

   (c) *Replacement policy*

   Changes relating to how pages are selected to be written out and replaced.

2. **Hoard implementation**

   Optimisations relating to the implementation of the hoard, which may be grouped as:

   (a) *Structure tuning*

   Changes to the size of address space, page table and d-page sizes.

   (b) *Interaction with the system*

   Changes related to reducing the overhead of system calls, and better integrating file I/O with the hoard.

The items listed under hoard policy will be considered in this chapter. Due to time constraints, those listed under hoard implementation are not addressed in this thesis and are included as part of the further work listed in Chapter 11.

Method

In the following sections, the implementation of each of the above changes will be considered, and the performance of the modified hoard will be compared with the times measured for the base configuration used in Chapter 8.

Each method will be examined separately from the others, the results analysed and compared, and likely high-performing combinations identified.

9.1 Write back only modified pages

This feature serves as a useful starting point when considering possible changes because it is relatively simple to implement and may be implemented independently of other changes. It will also serve as a building block for other changes examined in this chapter.

9.1.1 Overview

In a number of the benchmarks tested, there was a large margin between the amount of data read from main memory and that written back, particularly for the `179.art` and `181.mcf` programs. It may be possible to improve overall performance by writing less data to main memory by performing write-back of modified pages only.

9.1.2 Potential impact

The trade-off for this benchmark is between reducing the number of DMA *put* operations that are performed, against the cost of keeping a record of every write to hoard-managed memory locations.

To consider one particular example, the results for one of the runs of `181.mcf` from Section 8.5 are reproduced here in Figure 9.1. It is apparent that when using 2KB pages,

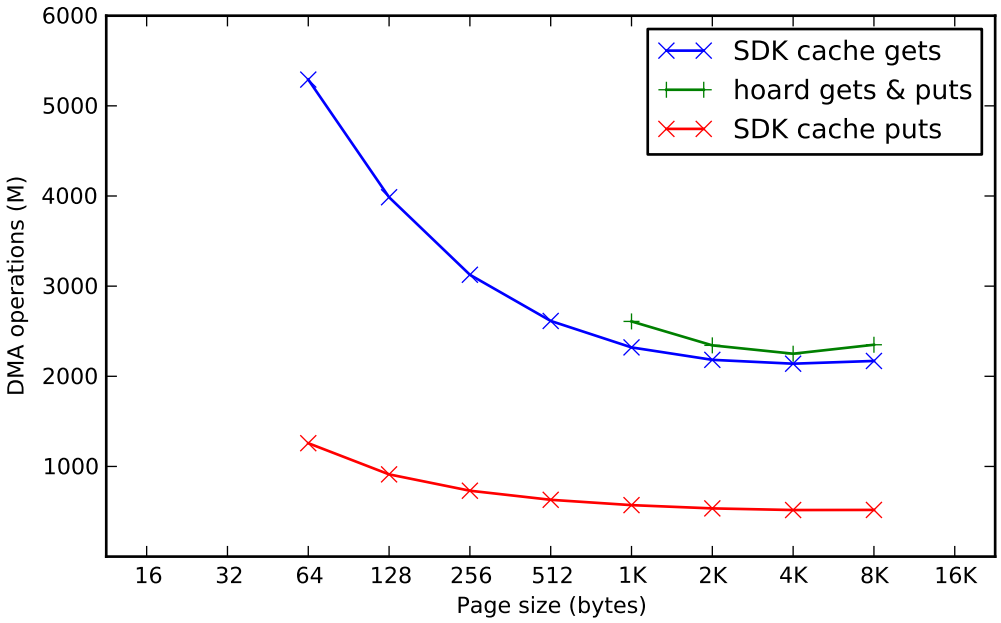


Figure 9.1: Split DMA *get* and *put* counts of hoard and SDK cache for `181.mcf`

greater than 1.8×10^9 more unmodified pages are written back to main memory by the hoard when compared with the SDK cache configuration.

Eliminating these 1,800 million writes should save a large amount of time. Based upon measurements using the dmabench application from the IBM SDK, a fenced, aligned 2KB page write will take approximately $110nS$ to complete, the amount of time saved can be estimated as $(1.8 \times 10^9\ writes) \times 110\text{ns} = 198\text{s}$, which would be a 12.2% reduction of the current runtime of 1,626 seconds.

9.1.3 Implementation

To implement this policy, a flag is required to indicate for each page whether it has been modified since being loaded. This style of operation does not map to a simple SPU instruction (or optimal sequence of instructions) and so some thought must be given to how this can be done efficiently.

Two methods are detailed below, each of which results in slightly different performance results.

9.1.3.1 Flagging the descriptor

Each loaded page has a descriptor that stores the meta-data for that page, and this descriptor serves as a useful place to store a dirty flag for the page.

A single bit is all that is needed, but it is more efficient to use a whole byte to encode the flag. Figure 9.2 shows the layout of a descriptor, containing a byte for the storage of page dirty-state.

Figure 9.2: One dirty byte, stored in the descriptor

The SPU instruction `fsmbi` (form select mask byte immediate) may be used to quickly generate a mask that can be logically or'd with the descriptor to set the dirty flag each time the page is written to. An example of this is in Figure 9.3.

This mask is used frequently throughout the program, and it would seem reasonable to keep the `fsmbi`-generated mask in one of the SPE's many registers to avoid the need to generate it more than once. In practice, the mask is generated at least once per function which does add a small extra latency.

```
fsmbi 0x0002 -> 00000000 00000000 00000000 0000ff00
descriptor   -> 0000c000 ffedc000 00000000 00010003
bitwise OR   -> 0000c000 ffedc000 00000000 0001ff03
```

Figure 9.3: Using the `fsmbi` instruction to mark a page dirty

9.1.3.2 Quadword-mask

Storing the modification flag in a page's descriptor requires extra steps to modify upon each write. Instead, a separate table for tracking the dirty state of pages may be used.

For this table, one machine word (16 bytes) is used to store the dirty flag for each page as the SPU instruction set does not include any instructions that permit writes to a region smaller than 16 bytes. Using this as the entry size ensures writes can occur with no preceding read.

The storage overhead of this method is substantial, and reduces the amount of local store space usable for hoard data. When using 1KB data pages, this table reduces the amount of space available for hoard pages by four.

In terms of performance, writes are able to be issued quickly and without dependence on other instructions, but do require the calculation of the target address from the page's local store address using a single shift instruction. In general, this extra step will be pipelined and add only a single extra cycle latency, but does require an extra register for the calculated address. This causes extra overhead when entering and leaving functions that dirty pages, as that extra register is stored to and loaded from the stack on every call.

9.1.4 Results

There is a small variation between each of the above approaches, with neither a clear winner on all of the benchmark programs. The trends are the same for both methods.

Program `qsort` is between 10% and 20% slower, depending on the exact configuration, where `hsort` is between 5% and 10% slower. These results relate to the high rate that pages are dirtied for each of these benchmarks — this change will increase the amount of work done for little or no benefit.

The `179.art` benchmark is around 5% faster in all configurations.

Every configuration of `181.mcf` was faster, from 6% to 23%. The case of 2KB pages

saw a speedup of almost 200 seconds in its fastest configuration, very close to the amount estimated earlier in this section.

Even with the fastest configuration, `181.mcf` is still 3% slower when using the hoard than it is with the fastest SDK cache configuration.

Runtimes for `183.equake` are between 3% slower and 3% faster for the smaller page size. The result the descriptor flag method with 8KB pages is faster due the generation of slightly smaller code, allowing one more hoard page to be available.

For most configurations, `mpeg2decode` is 3% to 5% slower. There is no result for the table method with 16KB pages for `mpeg2decode` as the program was not able to complete with the lesser number of pages available.

There is no data here for `julia_set` — it was not tested with this configuration as texture data it uses is not modified.

Table 9.1 illustrates the magnitude of DMA *put* operation reductions. For `179.art`, `181.mcf`, `183.equake` and `mpeg2decode` all see reductions of more than 50% of the total number of *puts* — more than 25% of all DMA operations. Apart from `mpeg2decode`, these programs do see runtime reductions, but a much smaller percentage.

9.1.5 Summary

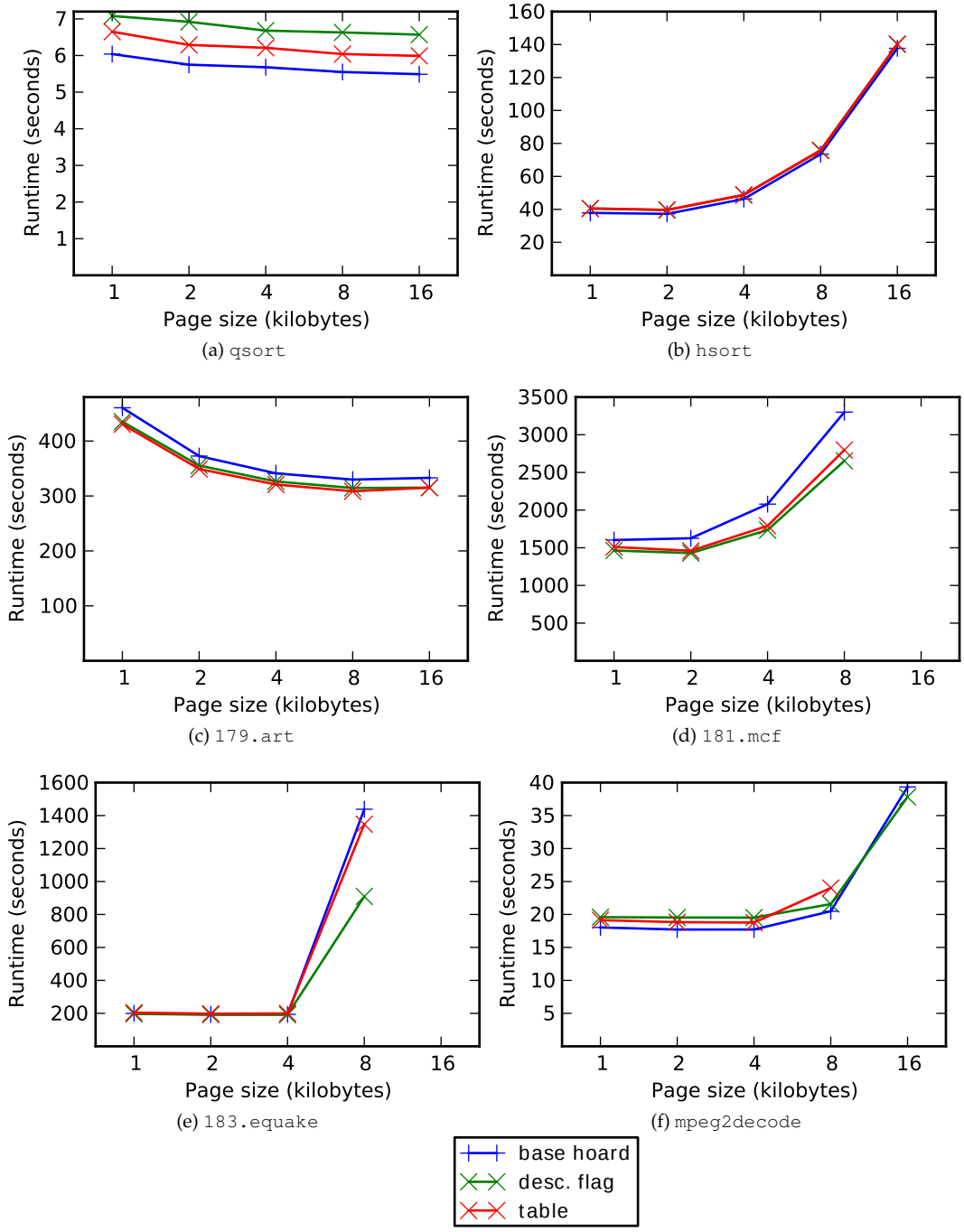
Writing only dirty pages degrades overall performance slightly in many cases, but results in performance improvements in others.

For programs that modify most loaded pages (`qsort`, `hsort`, `mpeg2decode`), there is no benefit to tracking page modification.

Both `179.art` and `181.mcf` modify approximately one page in five, and exhibit a notable performance improvement as a result.

Program `183.equake` modifies only one page in ten, but has only a two second runtime reduction. When the working set fits in the available hoard pages there are relatively few accesses to memory and little to be gained by reducing the already very low number of DMA operations.

The descriptor flag and table methods of tracking page modification perform similarly to one another, neither consistently outperforming the other. The table method does perform notably worse when the working set is too large for local store due to the reduced space available for pages.



(a) `qsort` (b) `hsort`

(c) `179.art` (d) `181.mcf`

(e) `183.equake` (f) `mpeg2decode`

Figure 9.4: Change in runtime for all tested programs when writing only modified pages

DMA *put* operations for `qsort`				
Page size	*Base*	*Modified only*	*Reduction*	*%*
1KB	1,235,523	1,240,349	-4,826	-0.39
2KB	613,879	616,904	-3,025	-0.49
4KB	308,018	309,305	-1,287	-0.42
8KB	157,512	157,492	20	0.01
16KB	81,728	81,719	9	0.01

DMA *put* operations for `hsort`				
Page size	*Base*	*Modified only*	*Reduction*	*%*
1KB	52,811,146	51,867,429	943,717	1.79
2KB	46,682,039	46,100,423	581,616	1.25
4KB	44,352,402	44,079,547	272,855	0.62
8KB	46,212,772	46,072,887	139,885	0.30
16KB	52,286,719	52,209,269	77,450	0.15

DMA *put* operations for `179.art`				
Page size	*Base*	*Modified only*	*Reduction*	*%*
1KB	615,284,607	106,498,184	508,786,423	82.69
2KB	309,850,942	55,033,862	254,817,080	82.24
4KB	159,137,775	29,394,125	129,743,650	81.53
8KB	85,154,411	17,098,096	68,056,315	79.92
16KB	49,551,907	12,491,416	37,060,491	74.79

DMA *put* operations for `181.mcf`				
Page size	*Base*	*Modified only*	*Reduction*	*%*
1KB	2,531,262,318	641,353,014	1,889,909,304	74.66
2KB	2,278,713,339	574,014,604	1,704,698,735	74.81
4KB	2,167,480,769	542,443,034	1,625,037,735	74.97
8KB	2,185,271,863	577,695,492	1,607,576,371	73.56

DMA *put* operations for `183.equake`				
Page size	*Base*	*Modified only*	*Reduction*	*%*
1KB	39,204,425	8,758,301	30,446,124	77.66
2KB	20,853,824	4,615,658	16,238,166	77.87
4KB	14,220,360	4,179,709	10,040,651	70.61
8KB	860,378,107	81,045,703	779,332,404	90.58

DMA *put* operations for `mpeg2decode`				
Page size	*Base*	*Modified only*	*Reduction*	*%*
1KB	488,702	197,326	291,376	59.62
2KB	250,451	102,354	148,097	59.13
4KB	154,454	70,151	84,303	54.58
8KB	2,260,147	2,497,726	-237,579	-10.51
16KB	10,194,244	12,683,537	-2,489,293	-24.42

Table 9.1: DMA *put* operations for base hoard configuration and descriptor flag method

9.2 Pre-writing pages

9.2.1 Overview

Improving the performance of any cache can be simply described as increasing hit rates and reducing the access times for hits and misses. When handling a miss, before loading the required page the page to be replaced must be written out to main memory. This introduces a sequential, blocking event that must complete before a new page may be loaded.

By pre-writing a page each time there is a miss and loading the next miss to this pre-written location, a large part of the overhead of DMA data transfer may be avoidable by allowing the program to run while a page is being written back to main memory.

It is anticipated that all configurations will benefit from this optimisation.

9.2.2 Potential impact

This change of policy should improve miss times, and little else. It will result in there being one less page available for caching of data, which is not expected to adversely affect overall program performance to a great degree.

9.2.3 Implementation

This requires only a small change to the hoard to implement, as the SPE MFC DMA ordering methods remain the same — a *put* is issued for a page and then a *get* is issued, fenced to occur after the *put* has completed to ensure data integrity. Rather than issuing the *get* immediately before the *put* as is the default case, the *put* of a subsequent page is issued after the fenced *get*. Figure 9.5 shows this change in the form of pseudo-code.

9.2.4 Results

Pre-writing pages provides a reduction in runtime in almost all cases, with greater improvements observed for smaller page sizes. Any performance benefit decreases as page size increases, with `mpeg2decode`, `179.art` and `hsort` showing an increased runtime with the largest page size.

For `179.art`, the 2KB page size configuration shows an improvement of 14% producing the fastest result when pre-writing pages, which is uncharacteristic as 2KB is one of the slower configurations for this program.

```
// No pre-write
page = find_page_to_replace()     // identify a page able to be replaced
put(page)                         // initiate DMA to main memory
get_fenced(page_requested, page)  // fetch desired page
sync(page)                        // wait until put and getf complete

// Pre-write
page = pre_written_page           // using page written during previous miss
get_fenced(page_requested, page)  // fetch desired page
pre_written_page = find_page_to_replace() // find another page
put(pre_written_page)             // initiate DMA to main memory
sync(page)                        // wait until request of new page completes
```

Figure 9.5: Page fetch logic change as pseudo-code

The best time for `181.mcf` is improved by almost 340 seconds (21%), which brings it to within 4% of the fastest SDK cache configuration.

Again, due to the relatively small amount of DMA traffic, `183.equake` is largely unchanged from the base hoard configuration. Runs are less than 1% faster with all page sizes.

`mpeg2decode` shows a reduction in performance as page size increases.

9.2.5 Summary

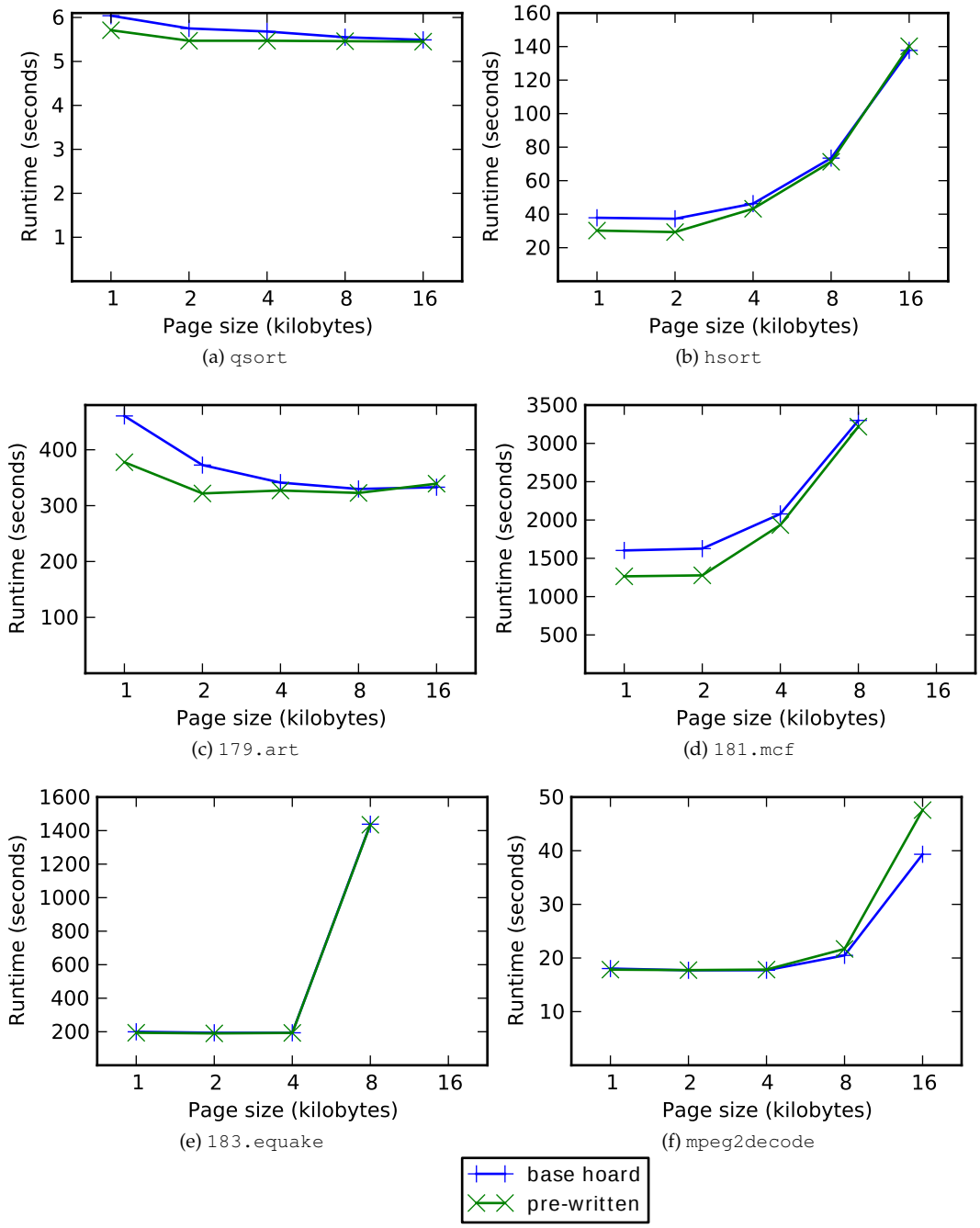
The results for this change indicate that it is generally beneficial, regardless of the program tested — with the exception of `mpeg2decode`, the shortest-running configuration runs faster when pre-writing pages.

As page size increases, the benefit decreases, turning into a time penalty in some cases. The reduced benefit for larger page sizes relates to the greater time required to complete the writing of a larger page to main memory, and that misses are happening at a rate higher than pages are able to be written back to main memory.

It may be possible to improve performance further by pre-writing more than one page, although it seems unlikely that this will help for large page sizes when this would reduce the capacity even more.

There is no provision in this implementation for 'recovering' a page that has been pre-written. If a page that has been pre-written is accessed, this is treated as a miss and that page is re-fetched from main memory, even though this is not required. An extra check before issuing the fetch would avoid this case, which may account for a measurable part of the performance decrease of larger page sizes.

By combining this method with the writing of modified pages only, it seems likely that time spent waiting for DMA operations would be reduced further.



(a) `qsort` (b) `hsort`

(c) `179.art` (d) `181.mcf`

(e) `183.equake` (f) `mpeg2decode`

Figure 9.6: Change in runtime for all tested programs when pre-writing

9.3 Write-through

9.3.1 Overview

A write-through policy may help improve performance by in many cases eliminating the delay of writing out a page before another can be fetched from main memory. The intended goal is to reduce the time taken to handle a miss.

9.3.2 Potential impact

The consequences of this change are less clear for up-front analysis as they will depend on the number of writes performed in a particular application and the comparative performance of extra, smaller writes against fewer larger ones.

It is anticipated that benchmarks that perform many writes, such as `qsort`, will be slowed by this change. Benchmarks that perform far fewer writes, such as `181.mcf` and `183.equake` may see performance improvements.

9.3.3 Implementation

This modification is again reasonably simple to implement for the hoard. A call to flush the modified memory line back to main memory is added for every operation that modifies hoard-managed data. This is a 128 byte write, naturally aligned.

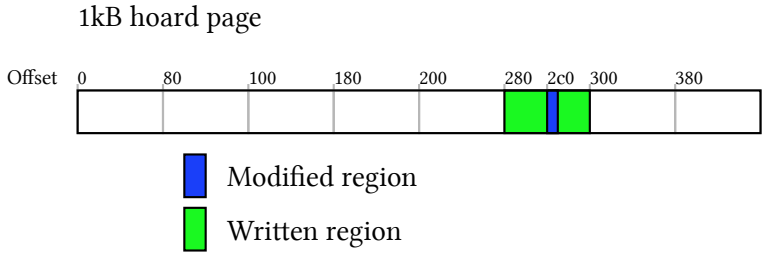


Figure 9.7: Region written of a modified page

The DMA tag ID used for the write of the line is that of the page containing the line. The advantage of this is that when a fenced DMA *get* is issued, the matching tag ID for the page will ensure safe ordering of the DMA operations.

9.3.4 Results

Again, there are no results for `julia_set` as it performs no writes.

A write-through policy clearly makes a difference, depending on the program and page size.

The shortest runtimes for `hsort`, `179.art`, `181.mcf` are made shorter by the use of a write-through policy, when compared to the base hoard configuration.

Write-through provides a greater benefit for worst-case, large page sizes, showing a reduction in the slowest configuration of 12% for `mpeg2decode`, 25% for `181.mcf`, 26% for `hsort` and 38% for `183.equake`.

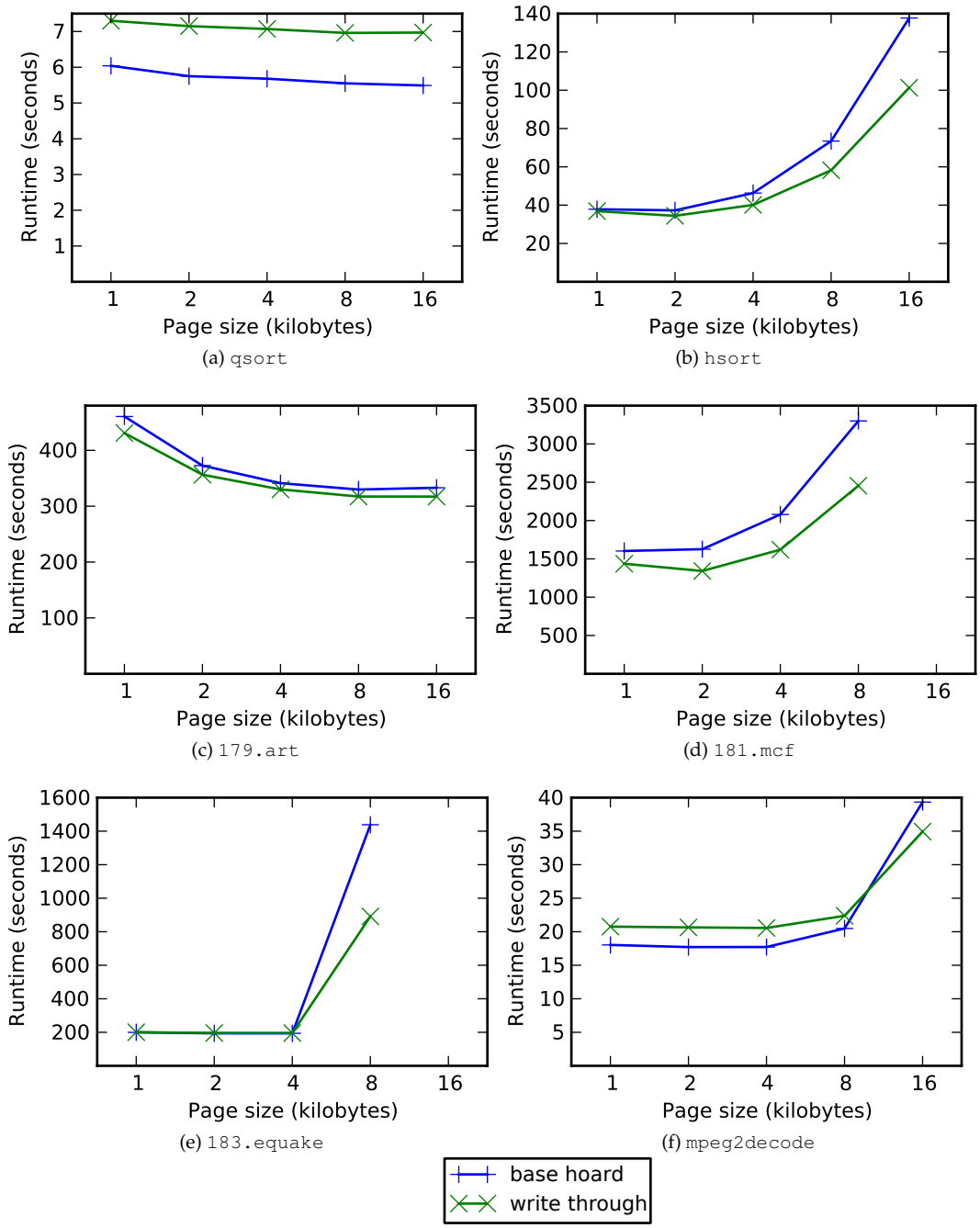
Otherwise, both `183.equake` and `mpeg2decode` are slower when using a write-through policy.

9.3.5 Summary

The usefulness of a write-through policy depends very much on the memory access patterns of a particular program.

It may be possible to further improve this method by selecting a page for replacement that has no outstanding DMA *put* awaiting completion.

To take advantage of spacial locality, it would be beneficial to be able to group writes in a way that would allow flushing the lines less frequently. In its current implementation, every access to a particular 128 byte region results in a DMA operation. Compile-time analysis might permit this kind of optimisation, but this is not achievable with the hoard implemented in its current form. It may be possible to gain some benefit with the addition of a single-entry memory-line address cache that forces a memory line write to memory only when writes change from one line to another.



(a) `qsort` (b) `hsort` (c) `179.art` (d) `181.mcf` (e) `183.equake` (f) `mpeg2decode`

Figure 9.8: Change in runtime for all tested programs when using a write-through policy

9.4 Pre-fetching data

9.4.1 Overview

For programs with predictable memory-access patterns, it may be possible to asynchronously fetch pages in advance, to hide some of the DMA latency inherent to a miss. The goal would be to increase cache performance by increasing the hit rate.

For this experiment, a single successor page will be pre-fetched for each page miss.

9.4.2 Potential impact

As well as increasing the amount of nearby data in local store, pre-fetching has the benefit that a load is issued in advance of a miss occurring and so may reduce the total amount of time that the program would spend waiting for large DMA transfers to be completed.

The pre-fetching of a successor page is quite a naïve method, making a simple, crude assumption about the program's memory access patterns. For programs that perform sequential, advancing, ordered accesses, this change should yield a performance increase. Programs that appear to have this behaviour include `179.art` and `183.equake`.

Another way to look at this change is to see it as effectively doubling page sizes without doubling the DMA transfer time overhead. Program `qsort` exhibits a trend that runtime decreases as page size increases, and it is anticipated that this change in policy will result in a faster runtime for that program.

The logic used for pre-fetching is shown in Figure 9.9.

```
load(X) {
    if(!prefetched(X)) {
        A = find_page_to_replace()
        put(A)
        get_fenced(X, A)
    }
    if(!loaded(Y) && !prefetched(Y)) {
        B = find_page_to_replace()
        put(B)
        get_fenced(Y, B)
        record_prefetch(Y)
    }
    sync(X)
}
```

Figure 9.9: Pre-fetch logic as pseudo-code

9.4.3 Implementation

This is a more intrusive modification to the program as it requires the hoard to have a concept of a page that is 'not fully loaded' into local store, and special handling is required as a result. Two methods were considered to implement this optimisation — the use of sentinel values stored in the lsa field of a page descriptor, and adding a separate field to store the local address of a pending operation.

9.4.3.1 Sentinel values in local-store addresses

Any access to hoard-managed data calls a hoard accessor function with the address of the data to be accessed. In the base implementation of the hoard, this function looks up the descriptor of the page requested and checks to see if the lsa field, in the primary slot, is non-zero. If it is non-zero, the data is already in the hoard, the actual address is calculated with the offset from the address passed to the function and the resultant location is returned. If it is zero, the miss handler is invoked and the data is loaded (as may be seen in Figure 9.10).

The first method attempted to handle pages that have been asynchronously pre-fetched

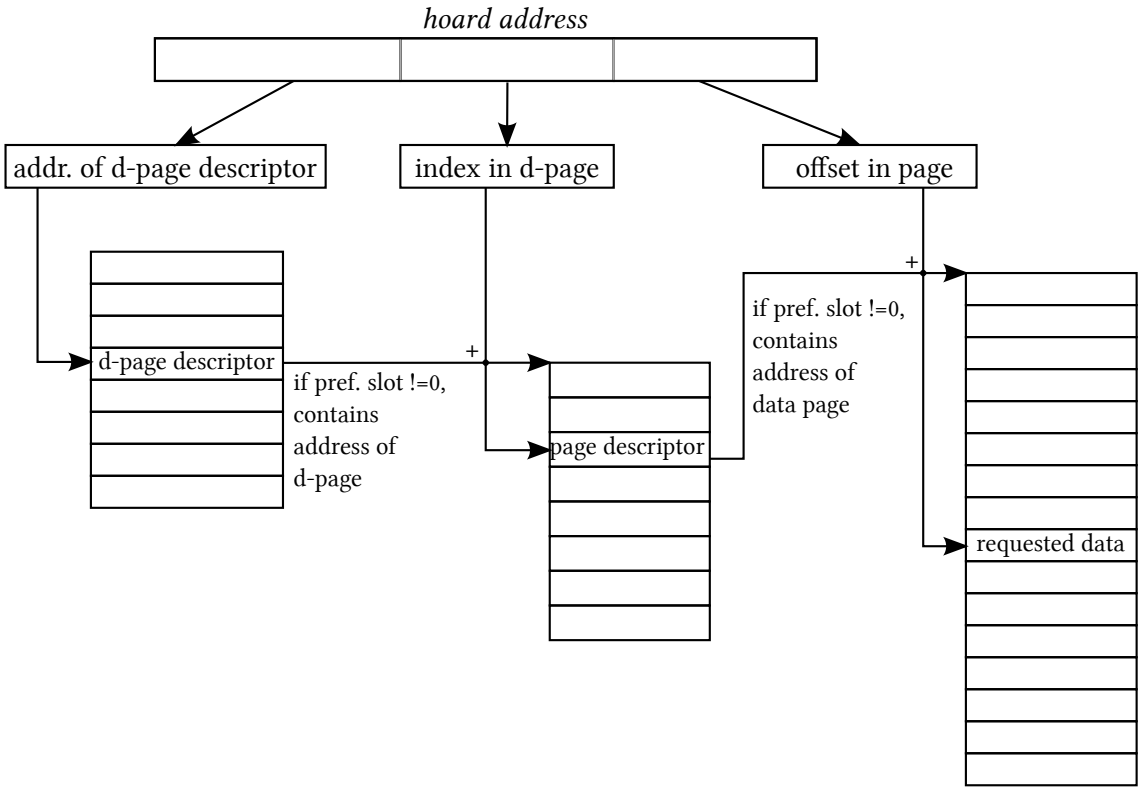


Figure 9.10: Regular hoard access

is to allow the storage of a non-address sentinel value in the lsa field of the descriptor.

The purpose of the sentinel is to indicate early-on that the access to the hoard is not a miss, but that it is also not a hit.

Figure 9.11 demonstrates the differences — particularly that rather than comparing against zero, the accessor function checks if the value is not a valid page address, assuming that there can be no valid pages stored in the range of memory locations from address 0 to the address of the first aligned page (i.e. `(page*)PageSize`)

If the value stored is a sentinel, an appropriate miss handler is invoked and the program will stall while the DMA operation for the page that has been pre-fetched completes.

The particular downside to this approach is that every memory access — read and write — is slowed a little by this extra check, and any extra cycle in this part of the hoard results in measurable runtime increases.

9.4.3.2 Separate pre-fetch flag in page descriptor

Rather than encoding the page number as a sentinel value, stored in the lsa field of the descriptor, it may instead be stored as an entirely separate field in the descriptor (illustrated

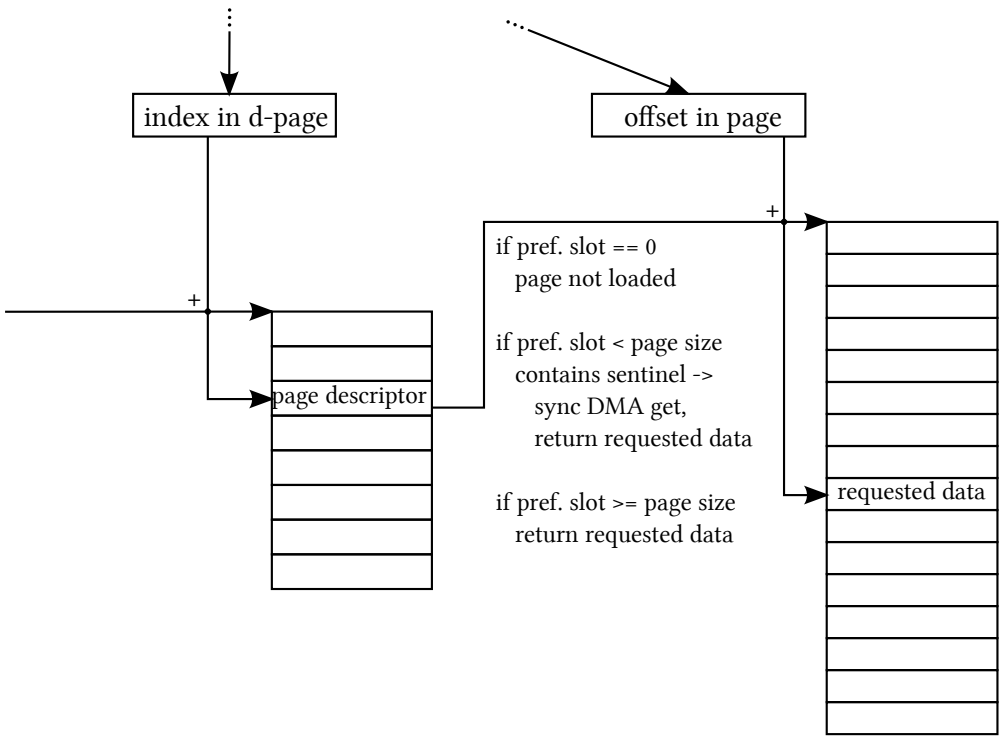
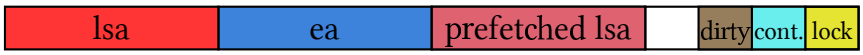


Figure 9.11: Hoard access, checking sentinel

in Figure 9.12). While it would be sufficient to store the hoard page number for a pre-fetched page, there is enough available space in the descriptor to store the full address of the page.

When the hoard accessor function examines the descriptor's lsa field and finds a zero (page not loaded, miss), the miss handler will be called. The first thing checked by the miss handler is the presence of a value in the pre-fetch field. If this is non-zero, the page has already been requested to local store, and all that need be done is to wait for the DMA operation to complete.

This requires a check for all misses, but does not slow down hits. This approach is the one that has been used to record the results presented here.



(if pre-fetched lsa is set, lsa must be zero)

Figure 9.12: Descriptor with pre-fetched page address

9.4.3.3 Remainder of the implementation

Pre-fetching a page assumes that the successor page contains related, relevant or interesting data — that which will be used by the program. In practice, memory allocations may have been performed in a fashion that does not necessarily enhance cache locality, and there is a finite end to the data that has been allocated.

The hoard manages memory accesses for the running program and it may only access memory that has been allocated directly by that program. This being the case, attempting to pre-fetching a successor page would be illegal if that page had not been allocated for use by the program.

To avoid invalid memory accesses, the hoard will never pre-fetch a page that is not described by the same d-page as the page that is being fetched. The advantage of this is that pre-fetching will never attempt to load a page that it does not have permission to access (the hoard's pooling memory allocator will ensure that all system heap allocations are at least the size describable by a single d-page) and a pre-fetch operation will never cause a d-page miss, which is a more complex case to handle.

Figure 9.14 shows the state of page descriptors in a d-page as a series of pages are requested, showing the change of state for each case. The first shows a page load and pre-fetch, the second shows a load of the last descriptor in the d-page with no pre-fetch, the

third shows a load where the successor is already loaded, and the fourth shows the load of an already pre-fetched page and the pre-fetching of its successor.

Pre-fetching introduces a DMA ordering problem that is not present in other methods tried up to this point and requires special care to handle correctly. This presented as a difficult-to-diagnose data corruption that occurred primarily with large pages when running cache-unfriendly workloads like `hsort` and `181.mcf`.

When pre-fetching a successor page, it is sometimes the case that a page that would be pre-fetched is the one that has been replaced by the miss that has just occurred — see Figure 9.13. In this case, the pre-fetch operation will attempt to reload the just-written page back into the hoard. If the data for this page is fetched from memory before the preceding write has completed, potentially old, stale data will be loaded into the hoard.

```
load(X) {
    ...
    put(A); get_fenced(X, A)
    ...
    put(B); get_fenced(Y, B) // B put - requires sync before reload
    ...
    sync(X)                  // synchronises put(A) and get(X)
}
...
load(B) {
    ...
    put(C); get_fenced(B, C) // error - reloading B with no sync
    ...
}
```

Figure 9.13: Fetching page written without sync as pseudo-code

DMA tags for transfers are selected based on the hoard slot to be used. The is no guarantee that a pre-fetched page will be synchronised before it is replaced as it may not be needed. The consequence of this is that special care is needed to ensure that a *put* preceding a pre-fetch is synchronised before the page that was *put* is fetched into another hoard page.

To address this problem, a record is kept of the address of the page that has been most recently written and not synchronised, along with the tag under which it was written. When fetching another page, a check is performed to ensure that a write of the page being fetched remains unsynchronised. If so, a DMA synchronisation operation is performed to ensure consistency. Figure 9.15 outlines this process.

i. Miss resulting in pre-fetch
Descriptor page:

x'0000	
x'1000	
x'2000	page not loaded - will be loaded
x'3000	page not loaded - will be pre-fetched
x'4000	
x'5000	
x'6000	
x'7000	

ii. Miss with no pre-fetch - end of dpage
Descriptor page:

x'0000	
x'1000	
x'2000	
x'3000	
x'4000	
x'5000	
x'6000	
x'7000	page not loaded - will be loaded

no subsequent pages in dpage - no pre-fetch

iii. Miss with no pre-fetch - loaded
Descriptor page:

x'0000	
x'1000	page not loaded - will be loaded
x'2000	page already loaded - no pre-fetch
x'3000	
x'4000	
x'5000	
x'6000	
x'7000	

iv. Miss - page pre-fetched
Descriptor page:

x'0000	
x'1000	
x'2000	
x'3000	page prefetched - load will be sync'd
x'4000	page not loaded - will be pre-fetched
x'5000	
x'6000	
x'7000	

Key:

Page not loaded
Page fully loaded
Page pre-fetched

Figure 9.14: Successor pre-fetching cases

```
// A record of any page that has not been synchronised
unsunc_page = 0
load(X) {
    ...
    if(X==unsunc_page) {
        unsunc_page = 0
        sync(X)
    }
    put(A); get_fenced(X, A)
    ...
    put(B); get_fenced(Y, B)
    if(unsunc_page!=0)
        sync(unsunc_page)
    unsunc_page = B
    ...
    sync(X)
}
```

Figure 9.15: Safe pre-fetching as pseudo-code

9.4.4 Results

A speedup of 5–10% is observed for `qsort`, depending on page size. In contrast, `hsort` runs 2–35% slower, as does `181.mcf`. For these programs with high miss rates, the extra loads make their problems worse.

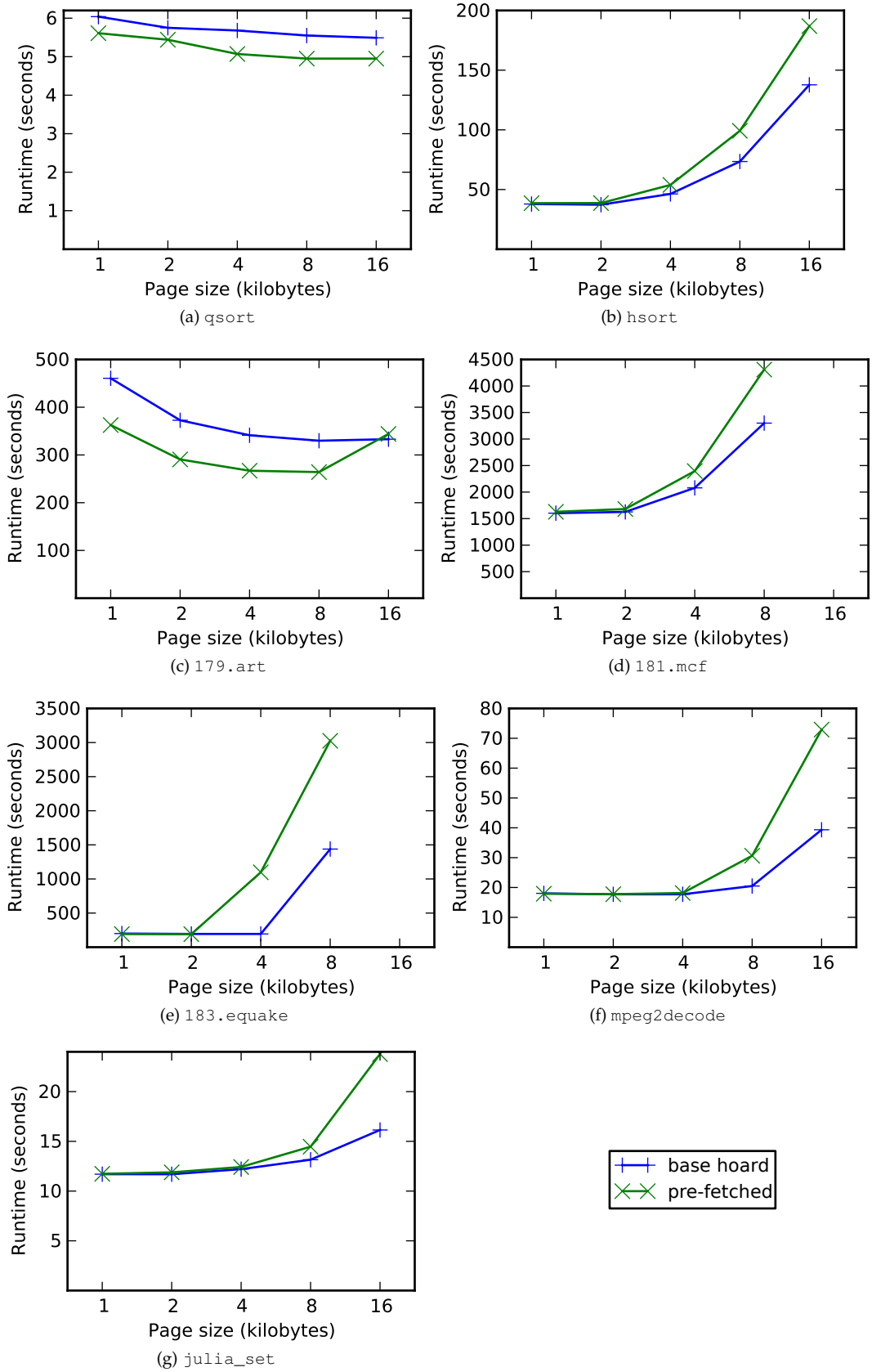
Program `179.art` demonstrates a speedup of 20–22% for all configurations except 16KB pages, which is likely related to the increased capacity pressure that pre-fetching introduces.

For `183.equake` a speedup of 3.5% and 1.5% is seen for 1 and 2KB page sizes respectively, and a large performance drop as the capacity pressure prevents the working set from remaining loaded.

Both `mpeg2decode` and `julia_set` have similar characteristics to `183.equake`.

9.4.5 Summary

In some cases, pre-fetching provides a large reduction in runtime. For the cases where it does not, the performance penalty is small for the respective best cases (<4%). The memory bandwidth of the Cell BE is such that even performing additional unhelpful memory transfers in this way incurs only a small overhead.



(a) `qsort` (b) `hsort`

(c) `179.art` (d) `181.mcf`

(e) `183.equake` (f) `mpeg2decode`

(g) `julia_set`

Figure 9.16: Change in runtime for all tested programs when pre-fetching a successor page

More specific tuning of pre-fetching to a program's specific access patterns could yield greater performance improvements. Specialisation of C++ accessor objects used to access the data could be one way to add different pre-fetch methods to the hoard. Such specialisations could be parameterised in a number of ways, including direction (successor, predecessor), stride (`a[i]`, `a[i+n]`), indirection of vectors of pointers (`*a[i]`, `*a[i+1]`, or `a[b[i]]`) or in other ways that match predictable memory access patterns (Chen et al. (2008b) addresses this type of problem).

In the case of `julia_set`, pre-fetching that took into account the tiling of texture data and the ordering of rendering patterns could reduce runtimes.

Initial investigation suggests that more advanced pre-fetching methods require complex changes in many parts of the hoard implementation, and depend heavily on the compiler to inline and perform other optimisations. The benefits of these methods are less clear, although the results of this section do suggest that further investigation is warranted.

9.5 Partial page fetching

9.5.1 Overview

Where pre-writing of pages and a write-through policy aim to reduce the time that must be spent waiting for write DMA to complete, partial page fetching seeks to reduce the fetch time delay that occurs when a full miss occurs. Rather than loading the whole page, the immediately required part of the page is loaded and the remainder is loaded in the background.

9.5.2 Potential impact

The goal of this change is a reduction in time spent waiting for fetches from main memory to complete. It will come with extra complexity and conditional cases throughout the miss handling code. This is likely to be a change that is sensitive to choice of page-part sizes and specifics of DMA ordering.

One approach to partial fetching will be tested in this section, that of fetching the required half directly and leaving the other half to be copied into local store asynchronously. This method appears to be on of the simplest ways to approach the problem partial fetching. Limited time prevented consideration of other approaches to partial page fetching.

It is not clear up-front how beneficial this method will be — smaller transfers are less efficient than larger ones, but should have lower latency.

9.5.3 Implementation

Like pre-fetching, this change requires the hoard to maintain a concept of not-completed 'pending' pages. In addition to tracking the hoard page that is being used to store the transfer, the portion(s) of a page that has been completed must also be recorded.

The balance to consider when implementing this change to hoard behaviour is that larger DMA transfers are more efficient, so while the initially requested portion may arrive sooner, the total time spent loading the page will likely be longer. It is also the case that dividing up the page into blocks causes a number of problems when it comes to ordering subsequent DMAs efficiently.

```
load(X, offset) {
    needed_half = (offset % PageSize) < HalfPageSize ? 0 : 1
    if(pending(X)) {
        if(pending_half(X) == needed_half) {
            // needed half not yet sync'd
            sync(X, half_bit)
        }
    } else {
        A = find_page_to_replace()
        if(pending(A)) {
            // write out the complete half
            put(A, !pending_half(A))
        } else {
            // page is not pending - write both halves
            put(A, 0)
            put(A, 1)
        }
        // fetch both halves
        get_fenced(X, A, 0)
        get_fenced(X, A, 1)
        // wait for needed half to complete
        sync(X, needed_half)
    }
    return lsa(X)
}
```

Figure 9.17: Split page fetching as pseudo-code

The approach adopted to trial this method is to split the fetching of each page into two parts — whenever a miss occurs, both halves are fetched and only the needed half is synchronised — an outline of the approach used is in Figure 9.17.

In the implementation tested, both page halves are fetched in the event of a miss. An alternate approach would be to fetch page halves on-demand. This may help reduce miss times but would not provide any pre-fetch advantage — the potential advantage of hiding the latency of the fetch of the other half behind program computation.

9.5.3.1 DMA ordering, again

Partial fetching experiences the same potential problem as pre-fetching — to the need to ensure the completion of a write to memory is complete before that effective address (*ea*) is re-read into an SPE's local store.

```
tmask = pmask // pmask is mask of pending DMA tags
tag = 0
ea = d->ea()  // ea of page to be loaded

// search for incomplete transfer
while(tmask) {
    if(tmask & 1) {
        // check for ea in eas - array of pending transfers
        if(eas[tag] == ea || eas[tag] == ea + HalfSize) {
            // if a match was found, wait for the DMA to complete
            mfcstat(tag)
            // remove it from the mask
            pmask &= ~(1 << tag)
            // leave the handler
            break
        }
    }
    // not found yet
    ++tag
    tmask >>= 1
}
```

Figure 9.18: Handling multiple pending, unfenced DMA operations as pseudo-code

The partial-fetch implementation used has a slightly more complex approach to avoiding out-of-order accesses. Where the pre-fetching method recorded only a single *ea* that had not been synchronised, the partial-fetch method maintains an entry for each of the 32 DMA tag IDs that may be used.

For each miss, the list of pending DMA operations must be scanned in its entirety. The MFC may be queried to provide a 32 bit mask of DMA tag IDs with incomplete transfers ongoing, and so it is possible to reduce the list of *ea*s to be compared against by using this mask. Figure 9.18 illustrates the method used.

9.5.4 Results

There is a small improvements for `179.art` of 5%.

Split page fetches do offer some advantage when using larger page sizes. `hsort`, `181.mcf` and `183.equake` each show a reduction in runtime with their largest tested page size of 6%, 20% and 16% respectively. `julia_set` also has reduced runtimes with larger page sizes.

9.5.5 Summary

Partial page fetching, as implemented here, does generally improve the runtime for the hoard with its largest page sizes. When using 1KB pages, it results in a small increase in runtime for all programs tested which reflects the increased inefficiency of performing smaller DMA transfers combined with the extra complexity involved in managing page state.

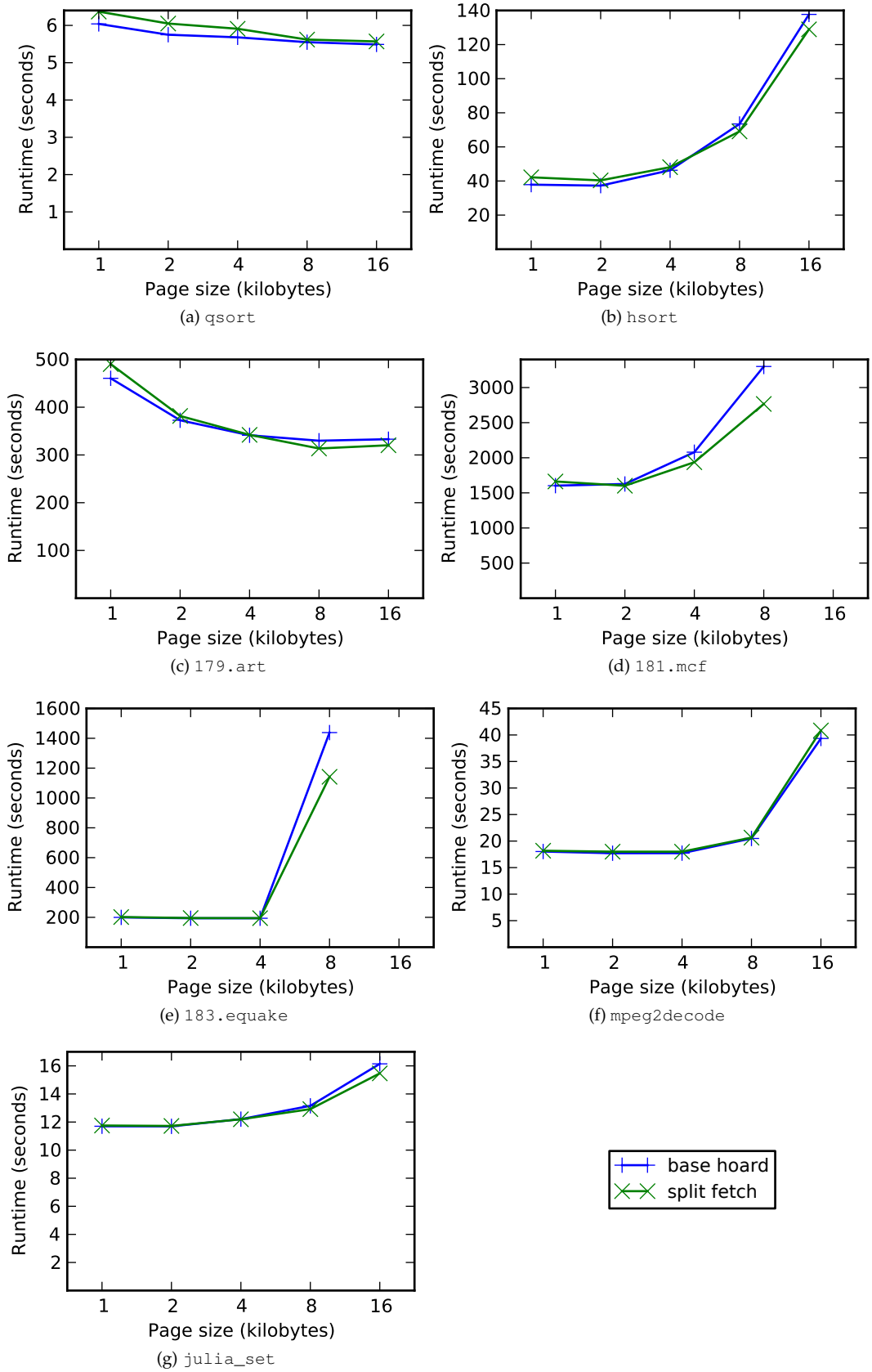
Further investigation into this method could include use of smaller page pieces, particularly when using larger page sizes.

9.6 Replacement policy

9.6.1 Overview

The hoard used a basic FIFO replacement policy that is simple to implement, fast to execute and requires no information about page usage, modification, or age. It is not an algorithm known for being particularly efficient for avoiding page replacements. The FIFO replacement algorithm was chosen as a simple starting point for the hoard. In this section, alternative, smarter replacement algorithms for the hoard will be examined.

(Note that Section 2.1.4 contains a summary of the page replacement methods mentioned in this section.)



(a) `qsort` (b) `hsort`

(c) `179.art` (d) `181.mcf`

(e) `183.equake` (f) `mpeg2decode`

(g) `julia_set`

Figure 9.19: Change in runtime for all tested programs when performing split page fetches

9.6.2 A simple alternative

The Second Chance replacement algorithm improves on FIFO through the use of a reference flag for each page which is set each time a page is accessed. When selecting a page to be replaced, the reference flag is checked. If set, the flag is cleared and the page moved to the end of the list of pages and the next page in the list is considered. The page is replaced if the flag is not set.

Pages that have been accessed since last they were considered are not replaced — they are given a second chance.

The Clock replacement algorithm is a form of Second Chance where a pointer (hand) to the next page to be replaced is used, rather than moving pages around in a queue. The hoard performs all page replacement selections through such a pointer. By adding a field to page descriptors to show that they have been referenced, and by giving pages a second chance when referenced, the Clock replacement algorithm has been implemented for the hoard.

This was implemented and the results are presenting in Figure 9.20, and it can be seen that adding a second chance does not provide a reduction in runtime — rather resulting in a small increase in most cases.

Table 9.2 presents the number of DMA *get* operations issued through the course of each benchmark. Clock reduces the total number of DMA operations in most cases, but not by enough to improve the overall runtime. A notable exception is `qsort` where the number of *gets* issued increases for most page sizes.

There is a decrease in DMA traffic of as much as 30% for `183.equake` with a relatively large increase in runtime, which are some of the longest runtimes for this benchmark with any hoard configuration.

There is very little change for `julia_set`. For the largest page size, there is a small but notable decrease in runtime, where the number of misses is reduced by more than 7%.

9.6.3 Observations

In some cases, Clock yields a very large reduction in DMA traffic but does not provide commensurate overall runtime reduction. To improve runtime through the use of better replacement algorithms, it will be necessary to reduce the amount of time spent accounting for accesses while also reducing the number of DMA operations.

Comparing the change in DMA operations for writing modified pages (in Table 9.1) with Clock (in Table 9.2), it is clear that tracking modified pages can be done in a way that often decreases runtime. Additionally, the performance of pre-writing, as presented in Section 9.2 suggests that combining these methods may be a way to achieve lower runtimes.

9.6.4 Alternate replacement algorithms

Based on these results and the optimisations already implemented for the hoard, four methods were devised and tested:

1. Second chance for modified pages, with pre-writing (2nd chance mod.)

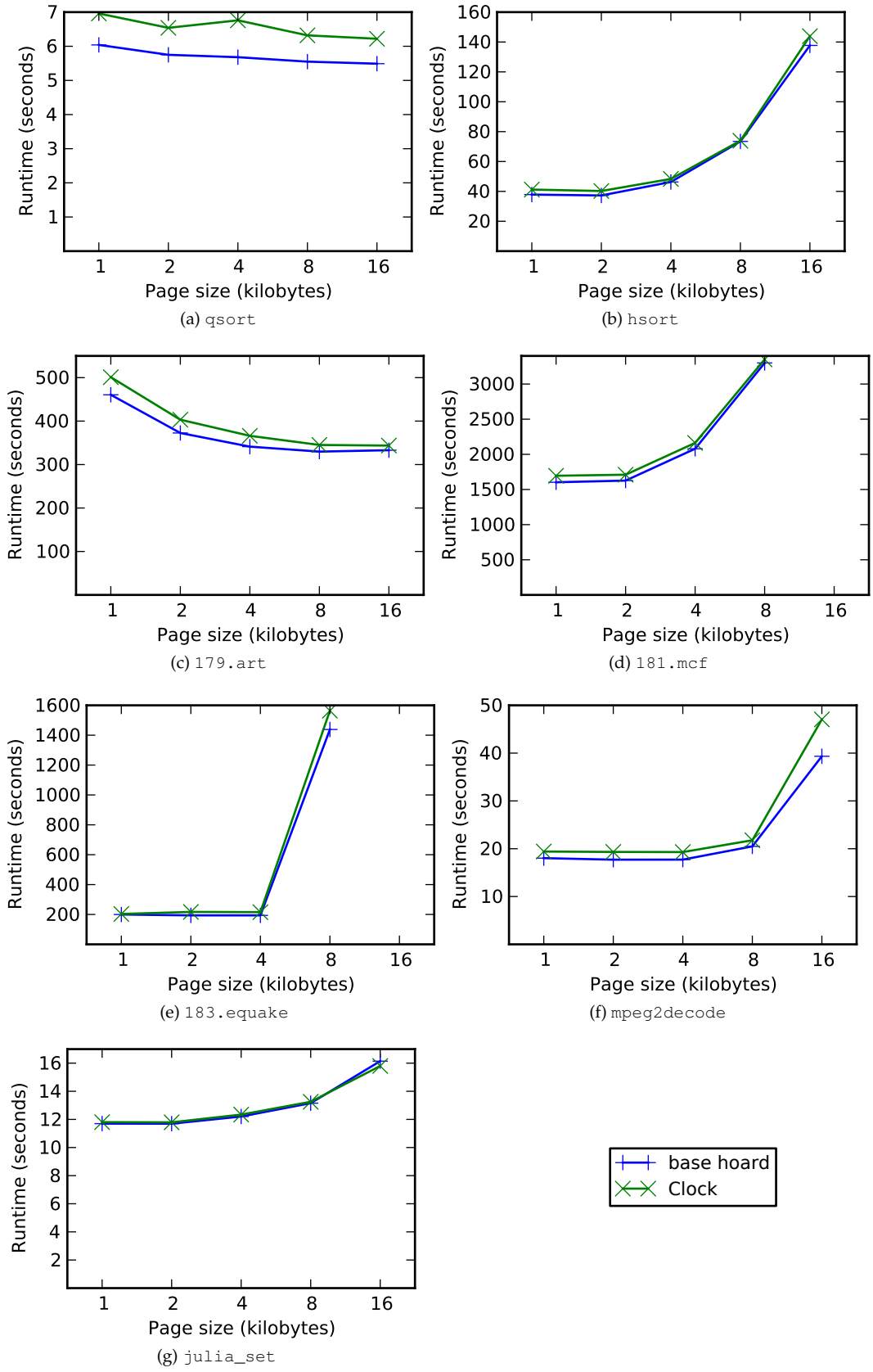
The results for Clock, as attempted above, show that the overhead of access accounting exceeds the benefit from reducing the number of page misses. Section 9.1 shows that writing back only modified pages is clearly beneficial to particular programs. By using the accounting required for this method to inform second chance decisions, there is no further overhead for tracking page modifications and pages are able to be written back to main memory in a way that has the potential to reduce DMA latency. In this way, fetch latency may be reduced, and pages may be effectively recovered, removing one of the penalties noted for pre-writing.

2. Pre-writing with a Least Recently Recovered ordering (LRR pre-written)

LIRS, Clock-Pro and other replacement algorithms (see Section 2.1.4) use a history of pages that are no longer stored in a cache to inform decisions about page replacement. In the case of the hoard, it is difficult to maintain a history of older pages that is searchable and accurate because of complexities of descriptor pages and their transient existence in limited local store.

Rather than attempting to maintain a long history of page use, the concept of pending pages was adopted from Section 9.4 to allow for there to be multiple pre-written pages with in flight DMA operations at one time. These pages are stored in a queue that captures their order of pre-writing.

The remaining resident pages are referenced in another queue. When a miss occurs, the new page is loaded to replace the least recent page in the pending queue, and a *put* of the the least recent page in the resident queue is issued.



(a) `qsort` (b) `hsort`

(c) `179.art` (d) `181.mcf`

(e) `183.equake` (f) `mpeg2decode`

(g) `julia_set`

Figure 9.20: Change in runtime for all tested programs when using Clock replacement

DMA *get* operations for `qsort`				
Page size	*FIFO*	*2nd Chance*	*Reduction*	*%*
1KB	1,235,523	1,260,579	−25,056	−2.03
2KB	613,879	627,389	−13,510	−2.20
4KB	308,018	314,166	−6,148	−2.00
8KB	157,512	159,531	−2,019	−1.28
16KB	81,728	81,537	191	0.23

DMA *get* operations for `hsort`				
Page size	*FIFO*	*2nd Chance*	*Reduction*	*%*
1KB	52,811,146	48,988,188	3,822,958	7.24
2KB	46,682,039	42,854,425	3,827,614	8.20
4KB	44,352,402	40,436,440	3,915,962	8.83
8KB	46,212,772	42,357,825	3,854,947	8.34
16KB	52,286,719	52,224,064	62,655	0.12

DMA *get* operations for `179.art`				
Page size	*FIFO*	*2nd Chance*	*Reduction*	*%*
1KB	615,284,607	611,476,880	3,807,727	0.62
2KB	309,850,942	306,316,150	3,534,792	1.14
4KB	159,137,775	153,556,725	5,581,050	3.51
8KB	85,154,411	77,265,331	7,889,080	9.26
16KB	49,551,907	42,526,153	7,025,754	14.18

DMA *get* operations for `181.mcf`				
Page size	*FIFO*	*2nd Chance*	*Reduction*	*%*
1KB	2,531,262,318	2,522,425,370	8,836,948	0.35
2KB	2,278,713,339	2,272,669,139	6,044,200	0.27
4KB	2,167,480,769	2,152,732,536	14,748,233	0.68
8KB	2,185,271,863	2,130,837,999	54,433,864	2.49

DMA *get* operations for `183.equake`				
Page size	*FIFO*	*2nd Chance*	*Reduction*	*%*
1KB	39,204,425	34,115,868	5,088,557	12.98
2KB	20,853,824	17,127,363	3,726,461	17.87
4KB	14,220,360	9,930,358	4,290,002	30.17
8KB	860,378,107	848,975,754	11,402,353	1.33

DMA *get* operations for `mpeg2decode`				
Page size	*FIFO*	*2nd Chance*	*Reduction*	*%*
1KB	488,702	483,568	5,134	1.05
2KB	250,451	244,514	5,937	2.37
4KB	154,454	131,153	23,301	15.09
8KB	2,260,147	2,053,402	206,745	9.15
16KB	10,194,244	12,683,537	−2,489,293	−24.42

DMA *get* operations for `julia_set`				
Page size	*FIFO*	*2nd Chance*	*Reduction*	*%*
1KB	14,512,774	14,439,992	72,782	0.50
2KB	13,317,230	13,195,771	121,459	0.91
4KB	14,428,049	14,254,992	173,057	1.20
8KB	14,339,487	14,010,882	328,605	2.29
16KB	15,462,669	14,332,906	1,129,763	7.31

Table 9.2: DMA *get* operations for FIFO and Second Chance Clock methods

The contents of the page are maintained in local store while the page is in the pending queue, so that if a page is referenced before it has been replaced it is able to be recovered and moved to the head of the pending queue (a constant time operation). This technique of ordering pages based on age and recovery from the pending queue will be referred to as Least Recently Recovered (LRR).

While queue manipulation introduces greater overhead, queues are manipulated only when a page miss occurs, which is much less frequently than for reads or writes from the hoard (the lowest hit rate recorded in this research is 85% for `181.mcf`).

This method does not track whether a page has been modified or not, and so pages will always be written back and may be written multiple times before being replaced, even if never modified.

3. Pre-writing of modified pages, with Least Recently Recovered ordering (LRR mod.)

To the previous method, this method adds tracking of page modification, only writing back pages that have been modified. The writeback of a page is issued when the page is moved from the resident queue to the pending queue, if it has been changed.

4. Second chance for modified pages, with pre-writing and Least Recently Recovered ordering (LRR 2nd chance)

This method combines methods 1 and 3, providing multiple page pre-writing, recovery, page modification tracking, page queues and a second chance for modified pages.

If the least recent page in the resident queue has not been modified, it is moved to the pending queue. If it has been modified, it is written back to memory and given a second chance.

9.6.5 Results

Note that the results for `mpeg2decode` with 16KB pages have been omitted for these replacement methods as they result in a very large increase in runtime or failed to execute correctly. The cause is the small number of pages available for regular use — zero or one.

Each of the tested methods will be considered in turn, discussing the timing results shown in Figure 9.21 and Table 9.3.

Second chance for modified pages (2nd chance mod.) provides benefits for most programs and page sizes. Performance scaling for this method appears to be consistent for all page sizes, maintaining a consistent margin from the base hoard configuration in most cases. It increases the runtime for `qsort` with all page sizes, while providing the best performance for `183.equake` with the largest page size.

Pre-writing with Least Recently Recovered ordering (LRR pre-written) produces the fastest runtime of these methods for `qsort`, `hsort` and `mpeg2decode` — the methods that write the most pages. For the others, it yields an improvement over the base hoard configuration, but this benefit tends to decrease as page sizes get larger. The reason for this is that this method may write more pages back to memory, and larger pages introduce greater latency. `julia_set` (which does not modify any pages) performs best with this method.

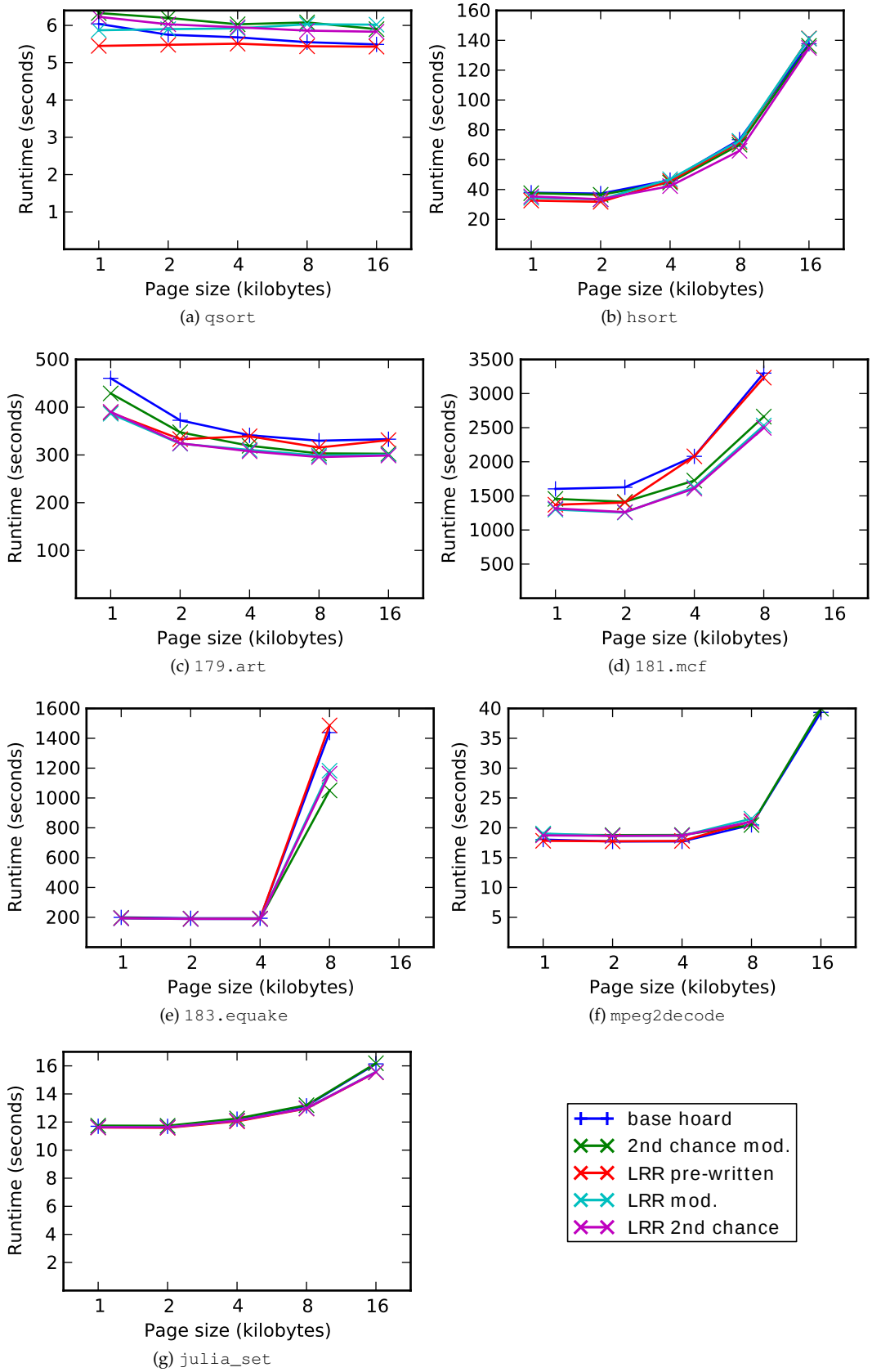
Pre-writing of modified pages, with Least Recently Recovered ordering (LRR mod.) shows better runtime performance for programs that modify fewer pages (`179.art`, `181.mcf` and `183.equake`) while causing programs that modify more pages to run slower than with the previous two replacement methods. Of the four algorithms tested, LRR mod. provides the fastest runtime for `181.mcf`.

Second chance for modified pages, with pre-writing and Least Recently Recovered ordering (LRR 2nd chance) shows very similar performance to the previous method, tending to have very slightly better performance with large page sizes and very slightly worse with smaller page sizes.

These final two methods show a slight reduction in performance over LRR pre-written due to the increased time spent accounting for writes that don't occur.

Second chance replacement may need to scan and write many pages to find one that has not been modified or recovered, and so is suitable to be moved to the pending queue. The SPU MFC supports a maximum number of sixteen DMA operations in-flight at any one time. When trying to issue more, program execution will be blocked until a transfer completes. The large number of smaller pages makes it more likely that DMA operations will block (see Section 4.2.2), causing greater delays.

Tables 9.4 and 9.5 show the reduction in DMA operations for each replacement method when compared to the equivalent base configuration, which reveal some details about the effectiveness of each method.



(a) `qsort` (b) `hsort`

(c) `179.art` (d) `181.mcf`

(e) `183.equake` (f) `mpeg2decode`

(g) `julia_set`

Figure 9.21: Change in runtime for all tested programs with various replacement methods

qsort

Page size	*1*	*2*	*4*	*8*	*16*
Base hoard	6.04	5.75	5.68	5.55	5.49
2nd chance mod.	6.33	6.20	6.03	6.08	5.90
LRR pre-written	5.45	5.48	5.51	5.44	5.43
LRR mod.	5.87	5.90	5.92	6.03	6.02
LRR 2nd chance	6.23	6.03	5.95	5.86	5.83

hsort

Page size	*1*	*2*	*4*	*8*	*16*
Base hoard	37.86	37.27	46.30	73.45	137.7
2nd chance mod.	37.54	36.46	44.69	70.02	136.26
LRR pre-written	32.60	31.76	45.66	71.75	141.32
LRR mod.	34.41	33.37	46.91	72.62	141.11
LRR 2nd chance	35.39	33.49	42.29	65.96	134.92

179.art

Page size	*1*	*2*	*4*	*8*	*16*
Base hoard	460.56	372.64	341.22	329.81	333.01
2nd chance mod.	428.93	347.48	319.18	303.23	302.55
LRR pre-written	388.51	333.14	339.09	315.53	331.00
LRR mod.	386.31	323.61	310.48	298.58	300.81
LRR 2nd chance	390.34	324.41	307.66	295.58	298.82

181.mcf

Page size	*1*	*2*	*4*	*8*	*16*
Base hoard	1,602	1,626	2,079	3,300	–
2nd chance mod.	1,456	1,412	1,724	2,662	–
LRR pre-written	1,370	1,402	2,079	3,233	–
LRR mod.	1,298	1,255	1,628	2,531	–
LRR 2nd chance	1,315	1,262	1,607	2,497	–

183.equake

Page size	*1*	*2*	*4*	*8*	*16*
Base hoard	199.61	193.78	193.76	1,438	–
2nd chance mod.	196.53	191.42	191.19	1,050	–
LRR pre-written	194.25	190.16	191.53	1,486	–
LRR mod.	193.20	189.32	188.92	1,182	–
LRR 2nd chance	193.49	189.31	188.89	1,162	–

mpeg2decode

Page size	*1*	*2*	*4*	*8*	*16*
Base hoard	18.03	17.71	17.72	20.49	39.34
2nd chance mod.	18.86	18.78	18.80	20.47	39.97
LRR pre-written	17.82	17.74	17.80	21.10	–
LRR mod.	19.04	18.65	18.68	21.52	–
LRR 2nd chance	18.73	18.65	18.69	21.02	–

julia_set

Page size	*1*	*2*	*4*	*8*	*16*
Base hoard	11.70	11.69	12.21	13.15	16.14
2nd chance mod.	11.75	11.73	12.25	13.2	16.19
LRR pre-written	11.61	11.59	12.05	12.96	15.54
LRR mod.	11.63	11.63	12.1	12.99	15.58
LRR 2nd chance	11.63	11.62	12.1	12.98	15.55

Table 9.3: Runtimes of all tested programs with various replacement methods

Table 9.4: Decrease in DMA operations with various replacement methods — `qsort`, `hsort` & `179.art`

Decrease in DMA operations for `qsort`

	2nd chance mod.			*LRR pre-written*			*LRR mod.*			*LRR 2nd chance*		
Page size	*gets*	*puts*	*total*	*gets*	*puts*	*total*	*gets*	*puts*	*total*	*gets*	*puts*	*total*
1KB	−2.10%	−6.98%	−4.54%	−0.26%	−0.47%	−0.36%	−0.26%	−0.47%	−0.36%	−2.49%	−7.71%	−5.10%
2KB	−2.20%	−7.20%	−4.70%	−0.40%	−0.96%	−0.68%	−0.40%	−0.96%	−0.68%	−2.69%	−8.32%	−5.50%
4KB	−2.00%	−7.33%	−4.66%	−0.72%	−1.83%	−1.28%	−0.72%	−1.83%	−1.28%	−2.52%	−9.45%	−5.98%
8KB	−1.28%	−7.79%	−4.54%	−0.15%	−2.49%	−1.32%	−0.15%	−2.49%	−1.32%	−1.60%	−12.00%	−6.80%
16KB	0.23%	−10.54%	−5.16%	0.30%	−8.28%	−3.99%	0.30%	−8.28%	−3.99%	−0.11%	−34.31%	−17.21%

Decrease in DMA operations for `hsort`

	2nd chance mod.			*LRR pre-written*			*LRR mod.*			*LRR 2nd chance*		
Page size	*gets*	*puts*	*total*	*gets*	*puts*	*total*	*gets*	*puts*	*total*	*gets*	*puts*	*total*
1KB	7.15%	−11.09%	−1.97%	0.68%	−1.63%	−0.48%	0.68%	0.91%	0.79%	6.36%	−13.86%	−3.75%
2KB	8.15%	−13.08%	−2.46%	1.79%	−0.61%	0.59%	1.79%	0.64%	1.21%	8.18%	−14.59%	−3.20%
4KB	8.80%	−17.18%	−4.19%	3.01%	−2.17%	0.42%	3.01%	−1.54%	0.74%	8.76%	−19.77%	−5.51%
8KB	8.31%	−17.61%	−4.65%	1.43%	−1.50%	−0.04%	1.43%	−1.20%	0.12%	11.67%	−20.96%	−4.65%
16KB	0.06%	−1.59%	−0.76%	0.46%	−3.13%	−1.33%	0.46%	−2.98%	−1.26%	0.47%	−16.21%	−7.87%

Decrease in DMA operations for `179.art`

	2nd chance mod.			*LRR pre-written*			*LRR mod.*			*LRR 2nd chance*		
Page Size	*gets*	*puts*	*total*	*gets*	*puts*	*total*	*gets*	*puts*	*total*	*gets*	*puts*	*total*
1KB	0.53%	82.78%	41.65%	0.62%	−0.02%	0.30%	0.62%	82.77%	41.69%	0.62%	82.76%	41.69%
2KB	0.84%	82.40%	41.62%	1.14%	−0.07%	0.54%	1.14%	82.36%	41.75%	1.14%	82.34%	41.74%
4KB	1.79%	81.80%	41.80%	3.51%	−0.20%	1.65%	3.51%	81.69%	42.60%	3.51%	81.62%	42.56%
8KB	4.92%	80.53%	42.72%	9.26%	−1.31%	3.98%	9.26%	79.92%	44.59%	9.26%	79.54%	44.40%
16KB	10.74%	77.10%	43.92%	21.16%	−78.93%	−28.88%	21.16%	66.78%	43.97%	21.16%	52.50%	36.83%

Table 9.5: Decrease in DMA operations with various replacement methods — `181.mcf`, `183.equake`, `mpeg2decode` & `julia_set`

Decrease in DMA operations for `181.mcf`

	2nd chance mod.			*LRR pre-written*			*LRR mod.*			*LRR 2nd chance*		
Page size	*gets*	*puts*	*total*	*gets*	*puts*	*total*	*gets*	*puts*	*total*	*gets*	*puts*	*total*
1KB	0.05%	74.82%	37.44%	0.67%	−0.89%	−0.11%	0.91%	75.25%	38.08%	0.74%	74.54%	37.64%
2KB	0.27%	74.66%	37.46%	0.62%	−0.83%	−0.10%	0.91%	75.29%	38.10%	0.77%	74.43%	37.60%
4KB	0.14%	74.75%	37.44%	1.17%	−1.33%	−0.08%	1.17%	75.50%	38.34%	1.37%	74.40%	37.88%
8KB	0.19%	74.85%	37.52%	2.37%	−5.99%	−1.81%	2.37%	73.56%	37.97%	2.66%	72.41%	37.54%

Decrease in DMA operations for `183.equake`

	2nd chance mod.			*LRR pre-written*			*LRR mod.*			*LRR 2nd chance*		
Page size	*gets*	*puts*	*total*	*gets*	*puts*	*total*	*gets*	*puts*	*total*	*gets*	*puts*	*total*
1KB	2.31%	82.21%	42.26%	10.29%	−2.36%	3.96%	11.76%	83.09%	47.42%	11.37%	80.99%	46.18%
2KB	2.04%	82.12%	42.08%	15.56%	−2.50%	6.53%	16.13%	82.70%	49.41%	16.30%	80.97%	48.63%
4KB	5.80%	80.01%	42.91%	30.79%	−12.51%	9.14%	33.19%	80.54%	56.86%	33.35%	77.25%	55.30%
8KB	5.98%	93.37%	49.68%	−18.07%	−28.72%	−23.39%	−17.47%	91.69%	37.11%	−12.79%	90.53%	38.87%

Decrease in DMA operations for `mpeg2decode`

	2nd chance mod.			*LRR pre-written*			*LRR mod.*			*LRR 2nd chance*		
Page Size	*gets*	*puts*	*total*	*gets*	*puts*	*total*	*gets*	*puts*	*total*	*gets*	*puts*	*total*
1KB	−2.10%	59.60%	28.75%	0.15%	−0.18%	−0.01%	0.29%	59.63%	29.96%	−1.92%	59.53%	28.80%
2KB	−4.60%	58.52%	26.96%	0.97%	−0.91%	0.03%	0.97%	59.36%	30.16%	−2.24%	56.64%	27.20%
4KB	−9.76%	55.09%	22.67%	13.57%	−25.97%	−6.20%	13.57%	54.58%	34.08%	8.11%	34.87%	21.49%
8KB	29.36%	39.15%	34.26%	20.05%	−241.12%	−110.54%	20.05%	−69.64%	−24.80%	30.38%	−123.47%	−46.54%

Decrease in DMA operations for `julia_set`

	2nd chance mod.			*LRR pre-written*			*LRR mod.*			*LRR 2nd chance*		
Page Size	*gets*	*puts*	*total*	*gets*	*puts*	*total*	*gets*	*puts*	*total*	*gets*	*puts*	*total*
1KB	-	-	-	0.23%	-	0.23%	0.23%	-	0.23%	0.06%	-	0.06%
2KB	-	-	-	0.48%	-	0.48%	0.48%	-	0.48%	0.37%	-	0.37%
4KB	-	-	-	0.89%	-	0.89%	0.89%	-	0.89%	0.84%	-	0.84%
8KB	-	-	-	2.26%	-	2.26%	2.26%	-	2.26%	2.26%	-	2.26%
16KB	-	-	-	7.75%	-	7.75%	7.75%	-	7.75%	7.75%	-	7.75%

All replacement methods cause an increase in the total DMA traffic for `qsort`, but LRR pre-written and LRR mod. do yield a reduction in runtime over the base configuration. These two also have the same effect on the number of DMA operations for `qsort` as LRR mod. behaves identically to LRR pre-written when every page is modified.

LRR pre-written is also the best performing configuration for `hsort`. As anticipated, LRR pre-written does show an increase in *puts* for every configuration.

The three LRR-based methods each reduce the number of misses for `179.art` by the same amount, with LRR 2nd chance being less effective for reducing writes than LRR mod., but still yielding the shortest runtime.

There is a large reduction in *puts* and a small reduction in *gets* for `181.mcf`.

Like `179.art`, `183.equake` has the shortest runtime when using the LRR 2nd chance method, even though this does not reduce the amount of DMA as much as LRR mod. The former does result in a slightly lower number of *gets*.

Program `mpeg2decode` performs the same or slightly worse in every configuration, making clear how little DMA traffic has to do with the overall runtime of this program.

9.6.6 Summary

All of the programs tested are able to have their runtime improved (or not adversely affected) through the use of at least one of these algorithms.

For programs that modify most of the pages they access, none of these methods is ideal as the accounting overhead is costly. The reduction in DMA traffic does not offset this.

For programs that modify fewer pages, smarter replacement methods are highly advantageous: `179.art` improved by 10%, `181.mcf` by 23%, and the runtime of `183.equake`, for which reductions are hard to achieve due to it's small amount of DMA activity, was reduced to the lowest of any method tried up to this point. These cases make clear that there are benefits possible from smarter replacement algorithms.

It is also clear that there is a fine line between expending processing effort to reduce DMA *gets* and *puts* rather than performing more, speculative transfers, and that the relative benefit depends very much on the memory access patterns of a particular program.

The replacement algorithms examined in this section have been tested on the basis of their simplicity of function, and their ease of implementation for the hoard, and are able to offer performance benefits. Small improvements can be expected through the implementa-

tion of other page replacement algorithms in a manner suited to the operation of the SPE, and this is something that is included in the further work of Chapter 11.

9.7 Optimisation conclusions

The optimisations trialled in this chapter aim to decrease total program runtime, and achieve this through a number of mechanisms.

9.7.1 Reducing write latency

Writing only modified pages, pre-writing pages, and using a write-though policy each aim to decrease the time spent waiting for writes to complete, but do so in different ways, and with different levels of success.

Writing only modified pages aims to transfer pages less frequently through the tracking of page modifications and achieves a goal of performing fewer DMA *put* operations. But it does not achieve better performance for all programs, unlike pre-writing.

Rather than performing fewer writes, pre-writing will typically cause a small increase in reads and writes to all programs as it reduces the number of available pages. The result, though, is a decrease in runtime for all programs, apart from `mpeg2decode`.

It is worth noting that none of the changes to the hoard tested in this chapter decreased the runtime of the `mpeg2decode` program; most caused negligible increases.

Using a write-through policy also attempts to reduce the time that the program must wait for *puts* to complete by issuing them immediately following a write to hoard managed data. While this did offer some benefit to some programs tested, it was not as great as was observed with pre-writing and caused large runtime increases to several programs due in part to the increase in total DMA traffic.

Of the three, pre-writing was the simplest to implement and yielded the greatest, consistent decrease in program runtime across all programs tested. The only program to record a faster runtime with an alternate method was `179.art` which showed slightly faster runtimes (<6.5%) with both of the other two methods.

9.7.2 Reducing fetch latency

Pre-fetching and partial-fetching both try to reduce the overhead of the DMA *get* component of a miss.

Pre-fetching seeks to fetch a desired page in advance of its use. Only a very simple successor page fetch was implemented in this section, with no close analysis of the programs in use, but this resulted in a large decrease in runtime for both `qsort` and `179.art`, both showing the fastest runtime of any configuration tested.

Pre-fetching incurred a small penalty on the best case for the other programs tested. The extra fetching of pages does not have a large DMA time cost, and small page configurations have a higher page count, meaning that erroneously fetched pages will not result in a significantly higher rate of conflict misses. Pre-fetching also has the characteristic that an unused pre-fetched page may be replaced straight away, offering a miss-optimisation similar in practice to pre-writing.

Partial fetching resulted in relatively small changes, with some notable benefit for larger page sizes.

Both these methods could be implemented together, although ensuring correct function and an efficient interaction between them would require a high degree of care.

9.7.3 Reducing the number of misses

While trying to decrease the time required to handle misses is useful, handling fewer misses could also be beneficial. The difficulty in doing so is that the accounting overhead needed to do so may take more time than that gained from performing fewer DMA operations, as evidenced when attempting to write only modified pages or when using the second chance Clock replacement algorithm.

Reducing miss rates is not enough on its own to reduce runtime, and so four simple replacement variants were tested to observe their overall performance. These make use of the write-latency reduction mentioned earlier in this section, and so their benefit should be considered in terms of reducing misses in addition to the others.

LRR pre-written provides the shortest runtime for `julia_set`, not because of any page pre-writing, but because pages are selected for replacement that works more efficiently with that program.

The LRR 2nd chance and LRR mod. methods produced very similar runtime results,

providing the fastest results for the `179.art`, `181.mcf` and `183.equake` benchmarks of all of the write-latency reducing methods for the hoard.

None of these methods does anything to reduce fetch latency, and it is likely that integrating these with a method like pre-fetching could further improve overall runtimes. Doing so would add further complexity to the process of handling misses.

The replacement methods trialled are not state of the art in the mechanisms that they use, and while it seems likely that hit rates could be improved with smarter methods, it seems unlikely that a substantial reduction in runtime could be attained.

9.7.4 Other possible improvements

Reducing hit times

For programs like `183.equake` and `mpeg2decode`, program runtime is dominated by the overhead of accessing the hoard. Miss rates are very low, and so much of the program's performance may be improved by reducing the time taken for the hoard to service a hit.

The current hoard page lookup routine is quite well optimised, but there is a great deal of potential to improve hit times by shrinking the size of the supported memory space. At the cost of some generality, it is possible to eliminate the need for descriptor pages, and so remove one level of indirection and testing from the lookup routine. Many of the programs tested in this research require an address space small enough that it needs only a first level descriptor table. Initial investigation shows that this change alone decreases the runtime of `179.art` by 10%.

There are techniques used in other SPE caching systems that would be worth trialling, particularly separate read and write indices, as used in COMIC (Lee et al. 2008), where written pages are referenced in a different page table. At the cost of 100% storage overhead, this can make tracking dirty pages as fast as read-only access.

Forcing the compiler to keep key offsets and values in registers could also help. Codeplay Offload does this (Codeplay Software Ltd. 2010b).

Reducing time to interact with other parts of the system

Many of the benchmarks used spend a time interacting with the system through the inefficient PPE callback mechanism, which is not well integrated with the hoard — system

library functions provide data transfer from file to hoard managed storage in a way that is somewhat convenient, but not very efficient.

Depending on the program, 10–30 seconds of measured runtime could be eliminated.

Using a 'better' compiler

Some initial testing with GCC 4.5 (released 14 April 2010) has shown that this newer version of the compiler will cause most of the programs used to run faster, from 5–30%. There is unfortunately not time to repeat all experiments using the newer version.

Chapter 10

Conclusions

This thesis explored the development of the hoard, a software paged virtual memory system for the Cell BE SPE, designed to work with the features and limitations of the platform to be straightforward and safe to use, simple to modify and to extend.

The Cell BE SPE is not a typical, general purpose multicore processor with homogeneous cores and automatic memory caches — instead its SPEs have local store memory and the need for explicit, asynchronous management of memory transfers performed by the running program.

This thesis demonstrated the usefulness of the hoard in software development for the SPE; that it simplifies the process of porting existing programs to the SPE and is able to deliver a high level of performance. The hoard facilitates the reuse of C and C++ programs or program fragments with minimal modifications.

10.1 Implementation

In this thesis, the hoard, a software paged virtual memory system for the Cell BE SPE, was compared to the software cache distributed as part of the IBM SDK for Multicore Acceleration Version 3.1.

The hoard design presented takes advantage of the SPE's features, providing fast access to cached program data through the use of table lookups. It provides full cache associativity, maximising the utilisation of local store while requiring as few cycles as possible to perform a lookup. Consequently, the hoard was shown to be able to handle hits faster than

the SDK cache, which has a structure more like a typical four-way set associative hardware cache.

The design of the hoard was shown to have some limitations when compared to the SDK cache — memory pages must be larger, and there are greater page table size overheads when using smaller hoard page sizes. Regardless, for most of the programs tested, use of the hoard resulted in substantially lower runtimes.

The hoard is a standalone software paged virtual memory system that provides a significant improvement in performance and usability over the SDK cache.

The hoard is implemented entirely as a compile- and run-time library, decoupled from the compiler. The use of C++ templates allow a high degree of type safety, and very few changes to existing code were needed to make use of the hoard (Appendix B contains a summary of the changes). This decoupling from the compiler avoided the need for a specialised toolchain to compile programs that make use of the hoard. It also allowed rapid iteration when developing and modifying the hoard itself as there is no need to rebuild the compiler.

Being implemented as a library did place limits on the amount that the hoard was able to interact with more complex characteristics of a programs behaviour. The limited static analysis prevented a number of more advanced program transformations from being attempted.

Writing and debugging the hoard proved to be a complex process. The lack of memory protection on the SPE made it easy to perform incorrect writes to memory, overwriting program code or data. Such memory corruption happened without warning and would result in program crashes, failed DMA operations and incorrect program results, sometimes long after the corruption has occurred.

The asynchronous operation of DMA transfers introduced further errors common in the presence of parallelism. The errors were often difficult to locate. Incorrectly ordered transfers resulted in runtime errors exposed only under certain conditions, such as high or low levels of traffic on the system's memory bus. Debugging was difficult in many cases as common tracing operations such as `printf()` change the timing of DMA operations.

The hoard relies on the compiler to perform aggressive inlining of hoard-related function calls, making debugging harder when working with optimised builds and runtime orders of magnitude slower when working with non-inlined debug builds.

10.2 Interface

The hoard utilises C++ templates and operator overloading to provide a 'natural' C-style and type-safe managed pointer implementation. This allowed data stored in main memory to be accessed from within an SPE program, and allowed the hoard to manage the transfer of data to and from local store in a manner transparent to the program. The hoard design discourages misuse of these managed pointers by ensuring that compile-time errors or warnings are generated for a range of incorrect or risky use cases.

Hoard managed pointers do not support every valid operation on a C pointer. They provide an effective tool, which works around some limitations of C++'s operator overloading and attempts to protect programmers from some of the riskier features of the C++ language. In the programs tested, unsupported pointer operations were able to be worked around with simple, functionally equivalent alternatives.

The hoard provides a less intrusive, more natural implementation of managed data access than the SDK cache. What it provides is closer in practice to named address spaces for C++ programs.

10.3 Performance

In the base configuration, examined in Chapter 8, the hoard showed significantly better performance than the SDK cache for most of the tested programs. Figure 10.1 shows that benchmarks `qsort`, `183.equake`, and `mpeg2decode` peak at more than 1.6 times faster than when using the SDK cache. The performance for these benchmarks reflects the high locality of memory access that each experiences, and shows the benefit of the faster lookup performed by the hoard.

In contrast, `181.mcf` demonstrated slower performance when using the hoard, as the runtime of this program is dominated by misses. The base configuration suffers particular penalties when running programs with relatively low locality or larger working sets due to the writeback of all pages and the larger minimum page size.

When using the minimum page size of 1KB, up to 80% of data page space may be lost to dynamic page table storage for particularly low-locality programs, negating any benefit that may have been gained through the use of a smaller page size.

In Chapter 9, methods to reduce the time required to handle page misses were pre-

sented. By reducing the time the program must wait for transfers to and from main memory, the general performance of the hoard was increased. This was achieved by performing DMA operations speculatively, and by reducing the number of transfers required.

Reducing the number of memory transfers did not automatically decrease program runtime. The bandwidth of the Cell BE memory architecture and the ability to asynchronously perform data transfers between the SPE and main memory allowed programs to speculatively read and write very large amounts of data without adversely affecting program performance.

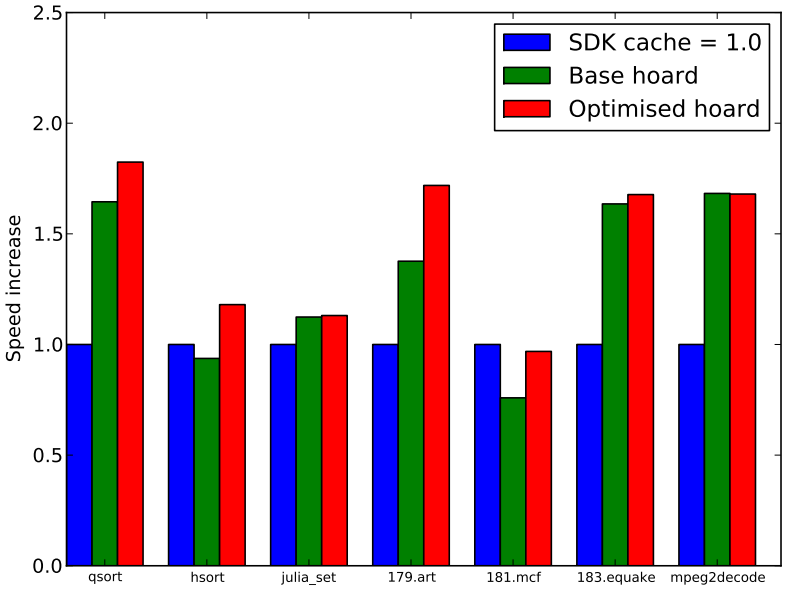


Figure 10.1: Speed increase of programs using the hoard over IBM SDK cache

The use of smarter replacement algorithms delivered better performance for some of the programs tested, although typically only by a small amount. While better replacement policies could be implemented for use with the hoard, it seems probable that they will not provide a large further decrease in runtime.

10.3.1 Comparative performance

It was difficult to provide a clear, meaningful comparison between programs running on the SPE with those running on other processors due to the diverse and complex architectural differences between systems. Comparing the runtime of the programs used with the same program running on the PPE does provide some interesting results. The design of the two types of core are very different internally but share a large amount of infrastructure on

the processor die and across the rest of the system.

Two of the programs that were tested ran faster on the SPE than on the PPE: `qsort` (for both hoard and SDK cache) and `179.art` (hoard only). *`179.art` ran more than 35% faster when using the hoard than it does on the PPE.*

For many of the other programs, performance was far worse on the SPE. The worst case measured was for `183.equake`, which ran four times slower when using the hoard (seven times slower using the SDK cache). The compiler was unable to eliminate repeat accesses to hoard-managed locations, and much of the inner-loop computation for this program required repeated lookups of the same array locations.

The hoard provides a great deal of value as a tool for porting programs to the SPE. While it offers good performance and is capable of better results through further refinement, its use provides a starting point from which further optimisation can be performed. For example, by manually caching array lookups in `183.equake`, the runtime of this program using the hoard can be reduced by more than half.

10.4 Optimal configuration

While it would be ideal to have a single configuration that offers the fastest possible speed for all programs, none of those tested achieves this goal. There are some hoard configurations that do offer consistently good results.

When using *page pre-writing* and a page size of 2KB, all of the tested programs showed improved or comparable performance to the base hoard configuration (Section 9.2).

The use of *pre-fetching* provided a very large performance boost to `qsort` and `179.art`, and with a 2KB page size caused only a very small runtime increase for the other tested programs (Section 9.4).

It seems likely that the use of these two methods with a page size of 2KB would provide high performance for a range of program memory access patterns. The actual performance of these methods combined was, unfortunately, not able to be tested due to shortage of time.

Pre-fetching and pre-writing are both blunt instruments — in the sense that they are applied unconditionally, without analysis of program behaviour. This is unlike other optimisations attempted that yielded less of a performance increase.

One of the beauties of a software caching system like the hoard is that it may be varied to suit the needs of a specific program, and may be extended, improved or removed depending on what will offer the best performance for a particular program.

Chapter 11

Further work

There are a number of further opportunities for improvement of the hoard.

11.1 Extending the scope of the hoard

The hoard manages only program data, but it may be interesting to consider its function in relation to the sort of software instruction cache outlined in Miller (2007) along with the function partitioning that has been proposed for the GCC compiler (Eres 2009) and the *soft-icache* work that has been done for the GNU binutils linker (Free Software Foundation 2010c), to see if it is possible to manage the code of larger programs in some useful fashion alongside data.

Dynamic code footprint optimisation, presented in Werth et al. (2009), seeks to reduce the amount of local store required for program code, potentially increasing space available for hoard pages.

Extending the hoard API to allow a programmer to specify caching policy on a per-hoard-pointer basis would provide finer-grained control for memory regions that have different access patterns. This could be exposed through C++ template specialisations. And would introduce a great deal of complexity to the implementation. The management of data with different policies creates more conditions that must be addressed by the hoard.

It may be more appropriate to provide functions that allow the programmer to annotate the program, providing hints or instructions to the hoard at appropriate points. Performance could be increased if the programmer was able to indicate explicit pre-fetches, page

flushes, and places in a program where many accesses will be to the same page.

A number of C++ standard library functions and containers could be optimised for hoard use, performing efficient pre-writing and -fetching, unchecked accesses where appropriate and more. Some initial prototyping in this area has shown promising results.

There would be benefit to teaching the compiler more about the relationship between cache access patterns and possible optimisations that could be applied to the generated program code, particularly with regard to constant (read-only) array access. In its current form, a full lookup must occur every time a hoard managed region of memory is accessed — the compiler will not keep a copy of a previously accessed location in a register unless it is explicitly stored by the program in a local variable. A clear example of this is the `smvp()` function of `183.equake`, where avoiding repeated multi-level array lookups by storing pointers in local variables can reduce total runtime by more than half.

It could be useful to add a profiling mode to the hoard which collects statistics for a program as it is run and then suggests a hoard configuration suited to the program.

11.2 Parallel computation

11.2.1 Test more parallel programs

Of the benchmarks considered, only `julia_set` made use of more than one SPE. This raises a question of how the hoard (and other comparable systems) will perform for other multi-core parallel programs. With more active cores, total memory bus contention would be expected to increase and there would be a consequent increase in latency when handling misses. Offsetting this, the memory bus is capable of higher sustained throughput when handling requests from multiple SPEs — consider the "Peak" line on Figure 8.17 compared to that for any other benchmark.

The likely change in performance will depend on the particular benchmark. For a program like `181.mcf` which exhibits a high miss rate, it seems probable that there would be performance degradation but that the extra latency would not dominate the runtime of each core — for 2KB pages, and when writing only modified pages, multiplying the measured throughput by six ($4.02 \times 6 = 24.12 GB/s$[1]) is less than the peak throughput measured ($25 GB/s$). If this is the case for a program with a very high miss rate, it would

[1] Based on SDK cache DMA traffic from Figure 8.27

seem that many other programs would not be penalised when running in parallel.

When writing only modified pages, the hoard has similar DMA throughput rates to the SDK cache. It seems likely that both would experience comparable performance degradation, if any.

11.2.2 Support for synchronisation

The hoard has no particular support for parallel access to hoard-managed memory locations from different processing cores. This is something that may be handled by the programmer directly, but there are some additions that would help improve performance.

To allow more advanced synchronisation between SPEs, storage of descriptor pages in main memory and the synchronisation of the first level page table between SPEs could help, as could communication between hoards on different SPEs using atomic memory accesses.

Invalidating the cache does need to be handled with care, and this is not a feature that has been considered in depth. When some region of memory has been modified by another process, it may be best to invalidate the entire cache and reload it on-demand, rather than selectively flushing or invalidating pages, as is suggested in Miller (2007).

11.3 COMIC

The COMIC system (Lee et al. 2008) appears to be a strong platform for developing high performance parallel programs for the Cell BE. A deeper comparison between the hoard and COMIC would be of some interest, but more useful would be a consideration of how the features and strengths of each could be combined — COMIC addresses many of the problems of parallel programming, where the hoard appears to have faster lookup and a simpler programming interface.

11.4 Further optimisation

There are a number of the optimisations from Chapter 9 that could be considered further.

In addition to the two methods for tracking dirty pages tested in Section 9.1, there may be a way to encode the dirty state of a page in the least significant bit of each 32 bit word of the descriptor, as this may be performed in all cases with a single ‘add word immediate’

(`ai`) instruction. For local addressing the lowest bit is ignored (addresses used for loads and stores are rounded down to a 16 byte boundary), and masking it out of the ea field for DMA operations is unlikely to have a measurable effect on performance.

The summary of Section 9.2 suggests that increasing the number of pre-written pages and recovering pre-written pages may be beneficial, and these are both achieved in the replacement algorithms tested in Section 9.6. There may be further benefit to pre-writing by dynamically varying the the number of pre-written pages based on whether the pre-write of the next page to be replaced has completed in time for the next page fetch.

A write-through policy (Section 9.3) shows benefits in some cases, but performs a large number of redundant DMA *puts* when adjacent writes are to the same memory line. A suggested extension to this method would be to maintain a record of the most recently written memory line, and only perform a DMA *put* if subsequent writes are to different parts of memory.

The pre-fetching method tested in Section 9.4 is a simple successor-only method, and it yields significant performance boosts for some tested programs. A range of different fetching methods could be trialled, taking into account strided access, indirect access and other dynamic methods. See Chen et al. (2008b), Liu et al. (2009) for other work in this area.

Fetching partial pages showed benefits in some cases in Section 9.5, and could be considered further with different sized pieces.

Section 9.6 showed that there are performance improvements achievable through the use of smarter replacement algorithms, and that further investigation into methods suitable for use on the SPE may be beneficial. Particularly, it would be interesting to see how an algorithm like Clock-Pro (Jiang et al. 2005) could be adapted to the SPE.

More work could be done to combine the methods tested to find greater improvements. Use of pre-writing and pre-fetching together appears likely to be beneficial.

D-pages are generated before the DMA operations for a page miss are initiated. It would be possible to hide some of the latency of a DMA operation (that is, hide the cost of d-page generation) by issuing the DMA operations first and generating the d-page while the DMA is under way.

Codeplay Offload(Codeplay Software Ltd. 2010b) keeps key addresses in registers throughout the program run, to minimise the time taken to look up data in its cache. If this technique can be achieved with the hoard, it could perhaps increase the performance of the

look up process.

11.5 Other programs

Programs and libraries that may benefit from hoard use include libz, libpng (particularly for PNG image decompression), libavcodec (especially for CABAC decoding of h.264 video streams), as well as for the Cell BE driver for the Mesa Gallium OpenGL implementation, which currently utilises the IBM SDK software cache.

Performance is able to be improved by removing the need for d-pages (the second level of lookup) for programs that address smaller amounts of memory. For example, programs `179.art`, `183.equake` and `julia_set` could be optimised further in this way. This was mentioned in Chapter 9.

Named address space support in GCC makes use of a software cache that functions similarly to the SDK cache. To be able to use the hoard in that context would be expected to provide similar potential for speedup as was observed in this research.

11.6 Closer investigation

Due to the page sizes used and the amount of local store available, the hoard is able to address only a small part of the virtual address space presented by the system, particularly when using smaller page sizes. This limitation could be removed or reduced by being able to maintain multiple first-level page tables in main memory, or to use a form of direct-mapping for the first (or second) level page tables. Implementation of such a method is likely to reduce performance, although it seems possible that such a method would still outperform the SDK cache.

11.7 Better hardware/OS integration

The Cell BE has a range of features that may be useful for further improving performance and working around limitations of the hoard. Explicit interaction with the SPE's TLB is possible, and there are a number of MFC features that may yield further advantages but have not been examined in full detail in this research.

An untested optimisation suggested in Chapter 9 is that of reducing the overhead of system and library calls that are handled by the PPE. To be able to more easily perform file and memory operations on effective addresses (rather than local store addresses) would remove the need to load large files in many small parts, which would eliminate many redundant memory transfers.

11.8 Optimisation of specific uses

The `julia_set` program performs texture lookups four at a time. Rather than blocking, to wait on the completion of each of the four before starting the next, it would be possible to optimise texture lookup by issuing asynchronous requests for all four and then synchronise at the time of use.

11.9 Other technological advances

11.9.1 Hardware

Programs `179.art` and `183.equake` work with double precision floating point datasets, but the Cell BE SPE experiences large penalties when working with double precision numbers — there is a six cycle stall every time a double precision instruction is executed.

Running these programs on a PowerXCell 8i system (e.g. the IBM QS22) would eliminate these stalls, but this hardware has a much slower memory architecture, utilising DDR2 in place of the XDR DRAM present in the QS20, QS21 and PlayStation 3 (IBM Corporation et al. 2009a).

These two programs have been run, in the base hoard configuration, on a QS22 system. While `183.equake` showed a small increase in performance (the runtime being still dominated by the number of accesses to hoard-managed data), `179.art` showed a small slow-down. This suggests that methods that were less beneficial when run on the Cell BE may be more beneficial on a system with a slower memory bus, or on a system running with higher load than was used for this research.

11.9.2 Software

Preliminary testing shows that GCC-4.5.0 offers some worthwhile performance increases for some of the benchmarks used in this thesis (up to 30% faster for `qsort`), but time constraints prevented recompiling and retesting all of the cases used.

Measuring and analysing the performance differences of this and other compilers. (such as XLC) would also be of interest.

Much of the process of converting a program to use the hoard is mechanical — replacing typenames with other typenames. The development of a program rewriting tool that helped speed up the process could be worthwhile. To be implemented as a plugin to an IDE such as Eclipse may also be readily achievable.

References

Acton, M 2007, *Introduction to SPU Shaders*, Insomniac Games, viewed 17 September 2010. <http://www.insomniacgames.com/tech/articles/0907/files/spu_shaders_introduction.pdf>

Adams, D 2002, *The Hitchhiker's Guide to the Galaxy — The Trilogy of Four*, Picador, London.

Adiletta, M, Rosenbluth, M, Bernstein, D, Wolrich, G & Wilkinson, H 2002, 'The Next Generation of Intel IXP Network Processors', *Intel Technology Journal*, vol. 6, no. 3, pp. 6–18.

Agarwal, A 1989, *Analysis of Cache Performance for Operating Systems and Multiprogramming*, Kluwer Academic Publishers, Norwell, MA, USA.

Ahmed, MF, Ammar, RA & Rajasekaran, S 2008, "SPENK: Adding another level of parallelism on the Cell Broadband Engine", in *IFMT '08: Proceedings of the 1st international forum on Next-generation multicore/manycore technologies*, ACM, New York, NY, USA.

Albrecht, T 2009, *Pitfalls of Object Oriented Programming — Game Connect Asia Pacific Conference, 2009*, Sony Computer Entertainment Europe, viewed 18 September 2010. <http://research.scee.net/presentations>

Atwood, J 2008, *Hardware is Cheap, Programmers are Expensive*, Jeff Atwood, viewed 20 July 2010. <http://www.codinghorror.com/blog/2008/12/hardware-is-cheap-programmers-are-expensive.html>

Balart, J, Gonzalez, M, Martorell, X, Ayguade, E, Sura, Z, Chen, T, Zhang, T, O'Brien, K & O'Brien, K 2008, 'A Novel Asynchronous Software Cache Implementation for the

Cell-BE Processor', in *Languages and Compilers for Parallel Computing: 20th International Workshop, LCPC 2007, Urbana, IL, USA, October 11-13, 2007, Revised Selected Papers*, Springer-Verlag, Berlin, Heidelberg, pp. 125–140.

Banakar, R, Steinke, S, Lee, BS, Balakrishnan, M & Marwedel, P 2002, 'Scratchpad memory: design alternative for cache on-chip memory in embedded systems', in *CODES '02: Proceedings of the tenth international symposium on Hardware/Software codesign*, ACM, New York, NY, USA, pp. 73–78.

Bansal, S & Modha, DS 2004, 'CAR: Clock with Adaptive Replacement', in *FAST '04: Proceedings of the 3rd USENIX Conference on File and Storage Technologies*, USENIX Association, Berkeley, CA, USA, pp. 187–200.

Baskaran, MM, Bondhugula, U, Krishnamoorthy, S, Ramanujam, J, Rountev, A & Sadayappan, P 2008, 'Automatic data movement and computation mapping for multi-level parallel architectures with explicitly managed memories', in *PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, ACM, New York, NY, USA.

Bellens, P, Perez, JM, Badia, RM & Labarta, J 2006, 'CellSs: a programming model for the Cell BE architecture', in *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, ACM, New York, NY, USA, p. 86.

Burks, AW, Goldstine, HH & von Neumann, J 1946, *Preliminary discussion of the logical design of an electronic computing instrument*, Deep Blue, viewed 17 September 2010. <http://hdl.handle.net/2027.42/3972>

Burroughs Corporation. Sales Technical Services 1961, *THE DESCRIPTOR: a definition of the B5000 Information Processing System*, Sales Technical Services, Equipment and Systems Marketing Division, Burroughs Corp.

Chen, T, Lin, H & Zhang, T 2008a, 'Orchestrating data transfer for the Cell/B.E. processor', in *ICS '08: Proceedings of the 22nd annual international conference on Supercomputing*, ACM, New York, NY, USA, pp. 289–298.

Chen, T, Zhang, T, Sura, Z & Tallada, M 2008b, 'Prefetching irregular references for software cache on Cell', *Proceedings of the sixth annual IEEE/ACM international symposium on Code generation and optimization*, pp. 155–164.

Choi, Y, Lin, Y, Chong, N, Mahlke, S & Mudge, T 2009, 'Stream Compilation for Real-Time Embedded Multicore Systems', in *CGO '09: Proceedings of the 7th annual IEEE/ACM International Symposium on Code Generation and Optimization*, IEEE Computer Society, Washington, DC, USA, pp. 210–220.

Codeplay Software Ltd. 2010a, *Codeplay Offload System for Cell Linux*, Codeplay Software Ltd., viewed 20 June 2010.
<http://offload.codeplay.com/download>

Codeplay Software Ltd. 2010b, *Offload: Community Edition*, Codeplay Software Ltd., viewed 18 September 2010.
<http://offload.codeplay.com>

compcache 2010, *Performance numbers for compcache*, compcache website, viewed 19 September 2010.
<http://code.google.com/p/compcache/wiki/Performance>

Debian 2010, *Debian — The Universal Operating System*, Debian, viewed 17 September 2010.
<http://www.debian.org>

Denning, PJ 1968, 'The working set model for program behavior', *Commun. ACM*, vol. 11, no. 5, pp. 323–333.

Denning, PJ 1970, 'Virtual Memory', *ACM Comput. Surv.*, vol. 2, no. 3, pp. 153–189.

Denning, PJ 1996, 'Virtual memory', *ACM Comput. Surv.*, vol. 28, no. 1, pp. 213–216.

Denning, PJ 2005, 'The locality principle', *Commun. ACM*, vol. 48, no. 7, pp. 19–24.

Dickey, TE 2010, *DIFFSTAT — make histogram from diff-output*, Thomas E. Dickey, viewed 21 September 2010.
<http://invisible-island.net/diffstat/>

Drepper, U 2007, *What Every Programmer Should Know About Memory*, Ulrich Drepper, viewed 29 October 2009.
<http://people.redhat.com/drepper/cpumemory.pdf>

Eichenberger, AE, O'Brien, JK, O'Brien, KM, Wu, P, Chen, T, Oden, PH, Prener, DA, Shepherd, JC, So, B, Sura, Z, Wang, A, Zhang, T, Zhao, P, Gschwind, MK, Archambault,

R, Gao, Y & Koo, R 2006, 'Using advanced compiler technology to exploit the performance of the Cell Broadband Engine architecture', *IBM Syst. J.*, vol. 45, no. 1, pp. 59–84.

Engel, J 2008, *Cache-efficient data structures, Kernel Miniconf, LCA2008*, video recording, distributed by Linux Australia.

Eres, R 2009, *[PATCH] New pass to partition single function into multiple (resubmission)*, mailing list, 2 June, gcc-patches, viewed 14 June 2009.
<http://gcc.gnu.org/ml/gcc-patches/2009-06/msg00166.html>

Fatahalian, K, Horn, DR, Knight, TJ, Leem, L, Houston, M, Park, JY, Erez, M, Ren, M, Aiken, A, Dally, WJ & Hanrahan, P 2006, 'Sequoia: programming the memory hierarchy', in *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, ACM, New York, NY, USA, p. 83.

Ferrer, R, González, M, Silla, F, Martorell, X & Ayguadé, E 2008, 'Evaluation of memory performance on the Cell BE with the SARC programming model', in *MEDEA '08: Proceedings of the 9th workshop on Memory performance*, ACM, New York, NY, USA, pp. 77–84.

Firefly Episode 1: The Train Job 2003, DVD, Twentieth Century Fox Film Corporation, USA, distributed by Twentieth Century Fox Home Entertainment South Pacific Pty Ltd. Directed by Joss Whedon.

Folding@home 2009, *Folding&home PS3 FAQ*, Stanford University, viewed 22 September 2010.
<http://folding.stanford.edu/English/FAQ-PS3>

Free Software Foundation 1999, *time — GNU Project*, Free Software Foundation, viewed 17 September 2010.
<http://www.gnu.org/software/time/>

Free Software Foundation 2010a, *Diffutils — GNU Project*, Free Software Foundation, viewed 21 September 2010.
<http://www.gnu.org/software/diffutils/>

Free Software Foundation 2010b, *GCC, the GNU Compiler Collection*, Free Software Foundation, viewed 17 September 2010.
<http://gcc.gnu.org/>

Free Software Foundation 2010c, *GNU Binutils*, Free Software Foundation, viewed 20 September 2010.
<http://www.gnu.org/software/binutils/>

Futurama Episode: The Birdbot of Ice-Catraz 2003, DVD, Twentieth Century Fox Film Corporation, USA, distributed by Twentieth Century Fox Home Entertainment South Pacific Pty Ltd. Directed by James Purdum.

Gonzàlez, M, Vujic, N, Martorell, X, Ayguadé, E, Eichenberger, AE, Chen, T, Sura, Z, Zhang, T, O'Brien, K & O'Brien, K 2008, 'Hybrid access-specific software cache techniques for the Cell BE architecture', in *PACT '08: Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, ACM, New York, NY, USA, pp. 292–302.

Goodman, JR 1983, 'Using cache memory to reduce processor-memory traffic', in *ISCA '83: Proceedings of the 10th annual international symposium on Computer architecture*, ACM, New York, NY, USA, pp. 124–131.

Gschwind, M, Hofstee, HP, Flachs, B, Hopkins, M, Watanabe, Y & Yamazaki, T 2006, 'Synergistic Processing in Cell's Multicore Architecture', *IEEE MICRO*, pp. 10–24.

Gupta, N 2010, *compcache - compressed in-memory swap device for Linux*, compcache website, viewed 19 September 2010.
<http://code.google.com/p/compcache/>

Hauck, EA & Dent, BA 1968, 'Burroughs' B6500/B7500 stack mechanism', in *AFIPS '68 (Spring): Proceedings of the April 30–May 2, 1968, spring joint computer conference*, ACM, New York, NY, USA, pp. 245–251.

Hennessy, JL & Patterson, DA 2007, *Computer Architecture, Fourth Edition: A Quantitative Approach*, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

Hofstee, HP 2005a, 'Power efficient processor architecture and the Cell processor', *11th International Symposium on High-Performance Computer Architecture, 2005. HPCA-11.*, pp. 258–262.

Hofstee, HP 2005b, *Power Efficient Processor Design and the Cell Processor*, Intern. Symp. on High Performance Computer Architecture, viewed 18 June 2008. <http://www.hpcaconf.org/hpca11/slides/Cell_Public_Hofstee.pdf>

Hoover, JN 2009, 'Air Force To Expand PlayStation-Based Supercomputer', *InformationWeek*, 20 November, viewed 17 September 2010. <http://www.informationweek.com/news/software/linux/showArticle.jhtml?articleID=221900487>

Horstmann, CS 2000, 'Memory management and smart pointers', in *More C++ Gems*, Cambridge University Press, New York, NY, USA, pp. 33–50.

Houston, M, Park, JY, Ren, M, Knight, T, Fatahalian, K, Aiken, A, Dally, W & Hanrahan, P 2008, 'A portable runtime interface for multi-level memory hierarchies', in *PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, ACM, New York, NY, USA, pp. 143–152.

IBM Corporation 1994, *The PowerPC(TM) Architecture: A Specification for a New Family of RISC Processors*, Morgan Kaufmann Publishers Inc., 2nd edn.

IBM Corporation 2007, *Cell Broadband Engine Linux Reference Implementation Application Binary Interface Specification Version 1.2*, Cell Broadband Engine Architecture Joint Software Reference Environment Series, IBM Systems and Technology Group.

IBM Corporation 2008a, *IBM XL C/C++ for Multicore Acceleration, V10.1 — Single-source compiler technology*, IBM Corporation, viewed 21 September 2010. <http://publib.boulder.ibm.com/infocenter/cellcomp/v101v121/topic/com.ibm.xlcpp101.cell.doc/getstart/ssc.html>

IBM Corporation 2008b, *Software Development Kit for Multicore Acceleration Version 3.1 — Accelerated Library Framework Programmer's Guide and API Reference*, IBM Corporation.

IBM Corporation 2008c, *Software Development Kit for Multicore Acceleration Version 3.1 — Data Communication and Synchronization Library Programmer's Guide and API Reference*, IBM Corporation.

IBM Corporation 2008d, *Software Development Kit for Multicore Acceleration Version 3.1 — Programmer's Guide*, IBM Corporation.

IBM Corporation, Sony Computer Entertainment Incorporated & Toshiba Corporation 2007, *Synergistic Processor Unit: Instruction Set Architecture Version 1.2*, IBM Systems and Technology Group.

IBM Corporation, Sony Computer Entertainment Incorporated & Toshiba Corporation 2008a, *C/C++ Language Extensions for Cell Broadband Engine Architecture Version 2.6*, Cell Broadband Engine Architecture Joint Software Reference Environment Series, IBM Systems and Technology Group.

IBM Corporation, Sony Computer Entertainment Incorporated & Toshiba Corporation 2008b, *Example Library API Reference*, IBM Systems and Technology Group.

IBM Corporation, Sony Computer Entertainment Incorporated & Toshiba Corporation 2008c, *Software Development Kit for Multicore Acceleration Version 3.1 — Programming Tutorial*, IBM Corporation.

IBM Corporation, Sony Computer Entertainment Incorporated & Toshiba Corporation 2008d, *SPE Runtime Management Library Version 2.3*, Cell Broadband Engine Architecture Joint Software Reference Environment Series, IBM Corporation.

IBM Corporation, Sony Computer Entertainment Incorporated & Toshiba Corporation 2008e, *SPU Application Binary Interface Specification Version 1.9*, Cell Broadband Engine Architecture Joint Software Reference Environment Series, IBM Systems and Technology Group.

IBM Corporation, Sony Computer Entertainment Incorporated & Toshiba Corporation 2009a, *Cell Broadband Engine: Programming Handbook (Including the PowerXCell 8i Processor) Version 1.12*, IBM Systems and Technology Group.

IBM Corporation, Sony Computer Entertainment Incorporated & Toshiba Corporation 2009b, *IBM SDK for Multicore Acceleration Version 3.1*, viewed 18 September 2010. <http://www.ibm.com/developerworks/power/cell/index.html>

ISO/IEC 1998, *Programming Languages — C++*, ISO/IEC 14882:1998(E), ISO/IEC.

ISO/IEC 2008, *Programming Languages — C — Extensions to support embedded processors*, ISO/IEC TR 18037:2008(E), ISO/IEC.

Jiang, S, Chen, F & Zhang, X 2005, 'CLOCK-Pro: an effective improvement of the CLOCK replacement', in *ATEC '05: Proceedings of the annual conference on USENIX Annual Technical Conference*, USENIX Association, Berkeley, CA, USA, pp. 35–35.

Jiang, S & Zhang, X 2002, 'LIRS: an efficient low inter-reference recency set replacement policy to improve buffer cache performance', *SIGMETRICS Perform. Eval. Rev.*, vol. 30, no. 1, pp. 31–42.

Johnson, T & Shasha, D 1994, '2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm', in *VLDB '94: Proceedings of the 20th International Conference on Very Large Data Bases*, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, pp. 439–450.

Karlsson, B 2004, *The Safe Bool Idiom*, Artima Developer, viewed 28 July 2010. <http://www.artima.com/cppsource/safebool.html>

Kerr, J 2009, *The SPU Filesystem, LCA2009*, Linux Australia, viewed 7 February 2010. <http://www.linux.org.au/conf/2009/slides/136.pdf>

Khanna, G 2010, *PlayStation 3 Gravity Grid*, University of Massachusetts Dartmouth, viewed 17 September 2010. <http://gravity.phy.umassd.edu/ps3.html>

Kilburn, T, Edwards, DBG, Lanigan, MJ & Sumner, FH 1962, 'One-level storage system', *IRE Transactions*, vol. EC-11, no. 2, pp. 223–235.

Kistler, M, Perrone, M & Petrini, F 2006, 'Cell Multiprocessor Communication Network: Built for Speed', *IEEE Micro*, vol. 26, no. 3, pp. 10–23.

Knight, TJ, Park, JY, Ren, M, Houston, M, Erez, M, Fatahalian, K, Aiken, A, Dally, WJ & Hanrahan, P 2007, 'Compilation for explicitly managed memory hierarchies', in *PPoPP '07: Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming*, ACM, New York, NY, USA, pp. 226–236.

Kudlur, M & Mahlke, S 2008, 'Orchestrating the execution of stream programs on multicore platforms', *SIGPLAN Not.*, vol. 43, no. 6, pp. 114–124.

Kunzman, DM & Kalé, LV 2009, 'Towards a framework for abstracting accelerators in parallel applications: experience with Cell', in *SC '09: Proceedings of the Conference on*

High Performance Computing Networking, Storage and Analysis, ACM, New York, NY, USA, pp. 1–12.

Kurzak, J, Buttari, A, Luszczek, P & Dongarra, J 2008, 'The PlayStation 3 for High-Performance Scientific Computing', *Computing in Science and Engineering*, vol. 10, no. 3, pp. 84–87.

Lee, C, Potkonjak, M & Mangione-Smith, WH 1997, 'MediaBench: a tool for evaluating and synthesizing multimedia and communicatons systems', in *MICRO 30: Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture*, IEEE Computer Society, Washington, DC, USA, pp. 330–335.

Lee, J, Seo, S, Kim, C, Kim, J, Chun, P, Sura, Z, Kim, J & Han, S 2008, 'COMIC: a coherent shared memory interface for Cell BE', in *PACT '08: Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, ACM, New York, NY, USA, pp. 303–314.

Levand, G 2008, *MARS — Multicore Application Runtime System, Linux Symposium 2008*, Sony Corporation of America, viewed 19 September 2010. <ftp://ftp.infradead.org/pub/Sony-PS3/mars/presentations/MARS-OLS-2008.pdf>

Liu, T, Lin, H, Chen, T, O'Brien, JK & Shao, L 2009, 'DBDB: Optimizing DMA Transfer for the Cell BE Architecture', in *ICS '09: Proceedings of the 23rd international conference on Supercomputing*, ACM, New York, NY, USA, pp. 36–45.

McCool, MD 2006, *Data-Parallel Programming on the Cell BE and the GPU using the RapidMind Development Platform*, CiteSeerX, viewed 2 July 2010. <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.126.8039&rep=rep1&type=pdf>

Megiddo, N & Modha, DS 2003, 'ARC: A Self-Tuning, Low Overhead Replacement Cache', in *FAST '03: Proceedings of the 2nd USENIX Conference on File and Storage Technologies*, USENIX Association, Berkeley, CA, USA, pp. 115–130.

Miller, J 2007, 'Software Instruction Caching', PhD thesis, Massachusetts Institute of Technology.

Moritz, CA, Frank, M & Amarasinghe, SP 2001, 'FlexCache: A Framework for Flexible Compiler Generated Data Caching', in *IMS '00: Revised Papers from the Second International Workshop on Intelligent Memory Systems*, Springer-Verlag, London, UK, pp. 135–146.

Mustain, AE 2005, *The Saga of the ARC Algorithm and Patent*, Varlena, LLC, viewed 26 July 2010.
<http://www.varlena.com/GeneralBits/96.php>

Neilforoshan, MR 2005, 'Synchronization and cache coherence in computer design', *J. Comput. Small Coll.*, vol. 21, no. 2, pp. 341–348.

Newlib 2010, *The Newlib Homepage*, Red Hat, Inc., viewed 17 September 2010.
<http://sourceware.org/newlib/>

Oberhumer, MF 2008, *LZO real-time data compression library*, oberhumer.com GmbH, viewed 23 September 2010.
<http://www.oberhumer.com/opensource/lzo/>

O'Neil, EJ, O'Neil, PE & Weikum, G 1993, 'The LRU-K page replacement algorithm for database disk buffering', *SIGMOD Rec.*, vol. 22, no. 2, pp. 297–306.

OpenMP Architecture Review Board 2008, *OpenMP Application Program Interface*, OpenMP Architecture Review Board, viewed 21 June 2010.
<http://www.openmp.org/mp-documents/spec30.pdf>

Organick, EI 1973, *Computer system organization: The B5700/B6700 series (ACM monograph series)*, Academic Press, Inc., Orlando, FL, USA.

Pankhurst, RJ 1968, 'Operating Systems: Program overlay techniques', *Commun. ACM*, vol. 11, no. 2, pp. 119–125.

Park, J, Lee, H, Hyun, S, Koh, K & Bahn, H 2009, 'A cost-aware page replacement algorithm for NAND flash based mobile embedded systems', in *EMSOFT '09: Proceedings of the seventh ACM international conference on Embedded software*, ACM, New York, NY, USA, pp. 315–324.

Park, Sy, Jung, D, Kang, Ju, Kim, Js & Lee, J 2006, 'CFLRU: a replacement algorithm for flash memory', in *CASES '06: Proceedings of the 2006 international conference on Compilers, architecture and synthesis for embedded systems*, ACM, New York, NY, USA, pp. 234–241.

Schneider, S, Yeom, JS, Rose, B, Linford, JC, Sandu, A & Nikolopoulos, DS 2009, 'A comparison of programming models for multiprocessors with explicitly managed memory hierarchies', *SIGPLAN Not.*, vol. 44, no. 4, pp. 131–140.

Seiler, L, Carmean, D, Sprangle, E, Forsyth, T, Abrash, M, Dubey, P, Junkins, S, Lake, A, Sugerman, J, Cavin, R, Espasa, R, Grochowski, E, Juan, T & Hanrahan, P 2008, 'Larrabee: a many-core x86 architecture for visual computing', *ACM Trans. Graph.*, vol. 27, no. 3, pp. 1–15.

Shippy, D & Phipps, M 2009, *The Race for a New Game Machine: Creating the Chips Inside the XBox 360 and the Playstation 3*, Citadel Press, New York.

Silberstein, M, Schuster, A, Geiger, D, Patney, A & Owens, JD 2008, 'Efficient computation of sum-products on GPUs through software-managed cache', in *ICS '08: Proceedings of the 22nd annual international conference on Supercomputing*, ACM, New York, NY, USA, pp. 309–318.

Standard Performance Evaluation Corporation 2005, *SPEC CPU2000 V1.3*, Standard Performance Evaluation Corporation, viewed 28 July 2010. <http://www.spec.org/cpu2000/>

Standard Performance Evaluation Corporation 2008, *SPEC CPU2006 System Requirements*, Standard Performance Evaluation Corporation, viewed 23 September 2010. <http://www.spec.org/cpu2006/Docs/system-requirements.html>

Stenström, P 1990, 'A Survey of Cache Coherence Schemes for Multiprocessors', *Computer*, vol. 23, no. 6, pp. 12–24.

Szydlowski, JD 2009, 'Federal officers use PlayStation 3 to combat child porn', *Scripps Howard News Service*, 17 November, viewed 17 September 2010. <http://www.scrippsnews.com/content/federal-investigators-use-playstation-3-combat-child-porn>

Tanenbaum, AS 2007, *Modern Operating Systems*, Prentice Hall Press, Upper Saddle River, NJ, USA, 3rd edn.

Taylor, MB, Kim, J, Miller, J, Wentzlaff, D, Ghodrat, F, Greenwald, B, Hoffman, H, Johnson, P, Lee, JW, Lee, W, Ma, A, Saraf, A, Seneski, M, Shnidman, N, Strumpen, V, Frank, M, Amarasinghe, S & Agarwal, A 2002, 'The Raw Microprocessor: A Computational Fabric for Software Circuits and General-Purpose Programs', *IEEE Micro*, vol. 22, no. 2, pp. 25–35.

Top 500 Supercomputer Sites 2010, *TOP500 List Releases*, TOP500.Org, visited 17 September 2010.
<http://www.top500.org/lists/>

Vasudevan, N & Edwards, SA 2009, 'Celling SHIM: compiling deterministic concurrency to a heterogeneous multicore', in *SAC '09: Proceedings of the 2009 ACM symposium on Applied Computing*, ACM, New York, NY, USA, pp. 1626–1631.

Weisstein, EW 2010, *Julia Set*, MathWorld–A Wolfram Web Resource.
<http://mathworld.wolfram.com/JuliaSet.html>

Wentzlaff, D, Griffin, P, Hoffmann, H, Bao, L, Edwards, B, Ramey, C, Mattina, M, Miao, CC, Brown, J & Agarwal, A 2007, 'On-Chip Interconnection Architecture of the Tile Processor', *IEEE Micro*, vol. 27, no. 5, pp. 15 –31.

Werth, T, Flossmann, T, Klemm, M, Schell, D, Weigand, U & Philippsen, M 2009, 'Dynamic code footprint optimization for the IBM Cell Broadband Engine', in *IWMSE '09: Proceedings of the 2009 ICSE Workshop on Multicore Software Engineering*, IEEE Computer Society, Washington, DC, USA, pp. 64–72.

Wilkes, MV 1965, 'Slave memories and dynamic storage allocation', *IEEE Transactions on Electronic Computers*, vol. EC-14, pp. 270–271.

Williams, S, Carter, J, Oliker, L, Shalf, J & Yelick, K 2008, 'Lattice Boltzmann simulation optimization on leading multicore platforms', in *Interational Conference on Parallel and Distributed Computing Systems (IPDPS)*.

Williams, S, Shalf, J, Oliker, L, Kamil, S, Husbands, P & Yelick, K 2006, 'The potential of the Cell processor for scientific computing', in *CF '06: Proceedings of the 3rd conference on Computing frontiers*, ACM, New York, NY, USA, pp. 9–20.

Wulf, WA & McKee, SA 1995, 'Hitting the memory wall: implications of the obvious', *SIGARCH Comput. Archit. News*, vol. 23, no. 1, pp. 20–24.

Zolman, L 2008, *STLFilt: An STL Error Message Decryptor for C++*, BD Software, viewed 20 September 2010.
<http://www.bdsoft.com/tools/stlfilt.html>

Appendix A

The hoard API

The hoard API is built around the `hoard_ptr` struct that works as a smart pointer — a drop-in replacement for many regular uses of a pointer.

Whenever a `hoard_ptr` is used for indirect access to the stored data, through dereferencing or array subscripting, a `hoard_ref` object is returned that ensures correct, safe access to and modification of hoard-managed data.

The interfaces of both classes are presented here. Their function and design rationale may be found in Chapter 6.

A.1 `hoard_ptr`

```
// hoard address typedef
typedef unsigned int ha_t;

// Macro to force function inlining when not debugging
#ifndef DEBUG
#define ALWAYS_INLINE __attribute__((always_inline))
#else
#define ALWAYS_INLINE
#endif

// null ptr type to prevent accidental conversions
struct null_ptr {
    null_ptr(null_ptr*) {}
};

template<typename T> struct hoard_ptr;

/* Specialisation - void* analog. Reduced functionality,
   stricter type controls */
template<> struct hoard_ptr<void> {
```

```
    // Simple accessor
    ha_t get_ha() const;

    // Constructors
    hoard_ptr();

    // Construct from a hoard address
    explicit hoard_ptr(ha_t ha_);

    // Construct from a different type -
    // allows silent cast to hoard_ptr<void>
    template<typename S>
    hoard_ptr(const hoard_ptr<S> o);

    // Permits silent 'upcast'. Yuck.
    // Wanted to be able to qualify this as explicit - C++0x feature.
    template<typename S>
    operator hoard_ptr<S>() const;

    // Safer conversion to bool
    // Lifted from boost/smart_ptr/detail/operator_bool.hpp
    operator unspecified_bool_type() const;

    // +ptr == ptr.  Surprise!
    hoard_ptr<void> operator+() const;
};

template<typename T> struct hoard_ptr {
    // simple accessor
    ha_t get_ha() const;

    // Constructors
    hoard_ptr();

    // copy
    hoard_ptr(const hoard_ptr& r);

    // construct with zero value
    hoard_ptr(assistor::null_ptr);

    // Construct from a ha.  Explicit to avoid accidental conversions
    explicit hoard_ptr(ha_t ha_);

    // Assign from zero
    hoard_ptr& operator=(assistor::null_ptr);

    // Assign from hoard_ptr
    hoard_ptr& operator=(const hoard_ptr& r);

    // unary +
    hoard_ptr operator+() const;

    // Safer conversion to bool
    // Lifted from boost/smart_ptr/detail/operator_bool.hpp
    operator unspecified_bool_type() const;

    // Compare with another hoard_ptr
    bool operator==(const hoard_ptr<T>& r) const;
```

```
    bool operator!=(const hoard_ptr<T>& r) const;
    bool operator< (const hoard_ptr &r) const;
    bool operator<=(const hoard_ptr &r) const;
    bool operator> (const hoard_ptr &r) const;
    bool operator>=(const hoard_ptr &r) const;

    // Arithmetic - hoard_ptr and integral offset
    hoard_ptr  operator+(size_t s) const;
    hoard_ptr  operator-(size_t s) const;
    hoard_ptr& operator+=(size_t s);
    hoard_ptr& operator-=(size_t s);
    hoard_ptr& operator++();
    hoard_ptr  operator++(int);
    hoard_ptr& operator--();
    hoard_ptr  operator--(int);

    // Arithmetic - comparing hoard_ptrs
    size_t operator-(const hoard_ptr& r) const;

    // Indirect accessors
    hoard_ref<T> operator[](int32_t i) const ALWAYS_INLINE;
    hoard_ref<T> operator*() const ALWAYS_INLINE;
    hoard_ref<T> operator->() const ALWAYS_INLINE;
};
```

A.2 hoard_ref

```
template<typename S> struct hoard_ref {
    explicit hoard_ref(uint ha_);

    // Conversion to referenced data
    operator const S&() const ALWAYS_INLINE;

    // Assignment from data
    hoard_ref& operator=(const S& r) ALWAYS_INLINE;
    // Assignment from another hoard_ref
    hoard_ref& operator=(const hoard_ref& r) ALWAYS_INLINE;

    // Various operations to permit hoard_ref to function as expected
    // Only those needed for tested programs
    S operator++(int) const ALWAYS_INLINE;
    S operator--(int) const ALWAYS_INLINE;
    hoard_ref& operator+=(const S& r) ALWAYS_INLINE;
    hoard_ref& operator-=(const S& r) ALWAYS_INLINE;
    hoard_ref& operator*=(const S& r) ALWAYS_INLINE;
    hoard_ref& operator|=(const S& r) ALWAYS_INLINE;
    hoard_ref& operator^=(const S& r) ALWAYS_INLINE {

    // comparison
    bool operator==(const S& r) ALWAYS_INLINE;
};

/* Specialisation for references to hoard_ptrs -
   analogous to pointer to pointer */
template<typename S> struct hoard_ref<hoard_ptr<S> > {
    explicit hoard_ref(uint ha_);
```

```
    // Assignment
    hoard_ref<hoard_ptr<S> >&
    operator=(const hoard_ptr<S>& r) ALWAYS_INLINE;
    const hoard_ref<hoard_ptr<S> >&
    operator=(const hoard_ptr<S>& r) const ALWAYS_INLINE;

    hoard_ref<hoard_ptr<S> >& operator=(uint r) ALWAYS_INLINE;
    hoard_ref<hoard_ptr<S> >&
    operator=(const hoard_ref<hoard_ptr<S> >& r) ALWAYS_INLINE;
    const hoard_ref<hoard_ptr<S> >&
    operator=(const hoard_ref<hoard_ptr<S> >& r) const ALWAYS_INLINE;

    // Indirect access
    hoard_ref<S> operator[](int32_t i) const ALWAYS_INLINE;
    hoard_ref<S> operator->() const ALWAYS_INLINE;

    // Comparison
    bool operator==(const hoard_ptr<S>& r) const ALWAYS_INLINE;
    bool operator!=(const hoard_ptr<S>& r) const ALWAYS_INLINE;
    bool operator==(assistor::null_ptr) const ALWAYS_INLINE;
    bool operator!=(assistor::null_ptr) const ALWAYS_INLINE;

    // 'pointer' arithmetic
    hoard_ptr<S> operator+(ptrdiff_t o) const ALWAYS_INLINE;

    // Conversion to stored data
    operator const hoard_ptr<S>&() const ALWAYS_INLINE;
};
```

Appendix B

Changes to benchmark programs

It was necessary to modify each benchmark to make use of the hoard. The following table summarises the number of lines that were inserted, deleted or modified in each source file for each benchmark tested.

As presented in Section 7.4, modifications can be categorised in the following ways:

- Conversion from K&R function syntax to C++ function syntax
- Renaming of variables that collide with C++ keywords
- Changing the types of variables and functions to hoard-managed types
- Explicit type conversions for vararg functions
- Replacing unsupported pointer uses with supported uses
- Addition of `hoard_ref` specialisation
- Type changes required by the C++ type system

The number of lines modified has been generated using `diffstat -m` (Dickey 2010) from a comparison of the original and modified files generated by the GNU `diff` program (Free Software Foundation 2010a). All line counts include white-space.

qsort

Source file	*Insertions*	*Deletions*	*Modifications*	*Total lines*
`data.h`	10	0	18	147
`qsort.c`	2	0	11	268
`qstack.h`	0	0	0	121
`util.h`	24	0	2	181
Total	36	0	31	717

hsort

Source file	*Insertions*	*Deletions*	*Modifications*	*Total lines*
`data.h`	10	0	18	147
`hsort.c`	12	0	9	222
`util.h`	26	0	2	181
Total	48	0	29	550

julia_set

Source file	*Insertions*	*Deletions*	*Modifications*	*Total lines*
`julia.h`	3	0	2	132
`state.h`	0	0	0	107
`pack16.h`	0	0	0	70
`ppm_util.h`	0	0	0	157
`julia.c`	0	0	0	891
`aos/ray.c`	0	0	0	432
`soa/ray.c`	5	0	6	1325
Total	8	0	8	3114

179.art

Source file	*Insertions*	*Deletions*	*Modifications*	*Total lines*
`scanner.c`	41	2	33	1270

181.mcf

Source file	*Insertions*	*Deletions*	*Modifications*	*Total lines*
defines.h	84	0	21	146
implicit.c	0	0	59	390
implicit.h	0	0	0	27
limits.h	0	0	0	24
mcf.c	0	0	0	145
mcf.h	0	0	0	30
mcfutil.c	3	0	42	389
mcfutil.h	0	0	0	31
output.c	0	0	7	81
output.h	0	0	0	25
pbeampp.c	3	0	6	224
pbeampp.h	0	0	1	25
pbla.c	3	0	7	75
pbla.h	0	0	2	26
pflowup.c	0	0	3	47
pflowup.h	0	0	1	25
prototyp.h	0	0	0	35
psimplex.c	0	0	11	146
psimplex.h	0	0	0	30
pstart.c	1	0	6	74
pstart.h	0	0	0	25
readmin.c	1	0	14	183
readmin.h	0	0	0	27
treeup.c	0	0	16	166
treeup.h	0	0	3	27
Total	95	0	199	2423

183.equake

Source file	*Insertions*	*Deletions*	*Modifications*	*Total lines*
quake.c	12	26	112	1513

mpeg2decode

Source file	*Insertions*	*Deletions*	*Modifications*	*Total lines*
config.h	0	0	0	44
display.c	0	0	0	1218
getbits.c	0	3	3	202
getblk.c	0	6	4	570
gethdr.c	0	1	1	1077
getpic.c	0	43	23	1225
getvlc.c	0	0	0	799
getvlc.h	0	0	0	491
global.h	0	0	10	488
idct.c	0	3	3	211
idctref.c	0	1	1	108
motion.c	0	17	6	236
mpeg2dec.c	0	8	14	767
mpeg2dec.h	0	0	0	129
recon.c	0	25	9	467
spatscal.c	0	18	18	331
store.c	0	22	28	576
subspic.c	0	21	11	392
systems.c	0	0	0	200
verify.c	0	0	0	303
Total	0	168	131	9834